ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO

CARLOS IVAN CASTRO MARQUEZ

ANÁLISE DA INFLUÊNCIA DO USO DE DOMÍNIOS DE
PARÂMETROS SOBRE A EFICIÊNCIA DA VERIFICAÇÃO
FUNCIONAL BASEADA EM ESTIMULAÇÃO ALEATÓRIA

São Paulo
2009

CARLOS IVAN CASTRO MARQUEZ

“ANÁLISE DA INFLUÊNCIA DO USO DE DOMÍNIOS DE
PARÂMETROS SOBRE A EFICIÊNCIA DA VERIFICAÇÃO
FUNCIONAL BASEADA EM ESTIMULAÇÃO ALEATÓRIA”

Dissertação apresentada à Escola Politécnica
da Universidade de São Paulo para obtenção
do Título de Mestre em Engenharia Elétrica.

São Paulo
2009

CARLOS IVAN CASTRO MARQUEZ

“ANÁLISE DA INFLUÊNCIA DO USO DE DOMÍNIOS DE
PARÂMETROS SOBRE A EFICIÊNCIA DA VERIFICAÇÃO
FUNCIONAL BASEADA EM ESTIMULAÇÃO ALEATÓRIA”

Dissertação apresentada à Escola Politécnica
da Universidade de São Paulo para obtenção
do Título de Mestre em Engenharia Elétrica.

Área de Concentração:
Microeletrônica

Orientador:
Prof. Dr. Wang Jiang Chau

São Paulo
2009

Este exemplar foi revisado e alterado em relação à versão original, sob responsabilidade única do autor e com anuência de seu orientador.

São Paulo, 4 de fevereiro de 2009.

Carlos Iván Castro Márquez

Prof. Dr. Wang Jiang Chau

FICHA CATALOGRÁFICA

Castro Márquez, Carlos Iván

Análise da influência do uso de domínios de parâmetros sobre a eficiência da verificação funcional baseada em estimulação aleatória, São Paulo, São Paulo, 2009. – Edição Revisada. 94 p.

Dissertação (Mestrado) - Escola Politécnica da Universidade de São Paulo, Departamento de Sistemas Eletrônicos.

1. Metodologias de Projeto 2. *System-on-Chip* 3. Verificação funcional. 4. Domínios de Parâmetros. 5. Análise de cobertura.

AGRADECIMENTOS

Um sincero agradecimento ao Professor Wang Jiang Chau, pelo seu constante apoio e dedicação. Ter aprendido estes anos com ele foi uma experiência inestimável.

A meu amigo Helbert Alonso Moreno, pelos constantes conselhos, suporte, camaradagem. Também pelas repreensões e solidariedade incondicional.

A minha mãe, meu pai, e meus irmãos, pelo ânimo, carinho, e ajuda constantes.

A todos meus colegas do Grupo de Sistemas Eletrônicos e Software Aplicado, que foram fonte de conhecimento e constante enriquecimento intelectual. Um especial agradecimento ao meu colega Edgar Leonardo Romero por toda sua ajuda e contribuições.

A Vasco Jerinić, e Jan Langer, da Universidade Técnica de Chemnitz, Alemanha, pelo apoio técnico que me ofereceram ao longo do desenvolvimento deste trabalho de mestrado.

À Escola Politécnica, e à CAPES, por todos os recursos e incentivos subministrados. Também, ao Brasil por disponibilizar as oportunidades que fazem possíveis este tipo de pesquisas.

RESUMO

Uma das maiores restrições que existe atualmente no fluxo de projeto de CIs é a necessidade de um ciclo menor de desenvolvimento. Devido às grandes dimensões dos sistemas atuais, é muito provável encontrar no projeto de blocos IP, erros ou *bugs* originados na passagem de uma dada especificação inicial para seus correspondentes modelos de descrição de hardware. Isto faz com que seja necessário verificar tais modelos para garantir aplicações cem por cento funcionais.

Uma das técnicas de verificação que tem adquirido bastante popularidade recentemente é a verificação funcional, uma vez que é uma alternativa que ajuda a manter baixos custos de validação dos modelos HDL ao longo do projeto completo do circuito. Na verificação funcional, que está baseada em ambientes de simulação, a funcionalidade completa (ou relevante) do modelo é explorada, aplicando-se casos de teste, um após o outro. Isto permite examinar o modelo em todas as seqüências e combinações de entradas desejadas. Na verificação funcional, existe a possibilidade de simular o modelo estimulando-o com casos de teste aleatórios, o qual ajuda a cobrir um amplo número de estados.

Para facilitar a aplicação de estímulos em simulação de circuitos, é comum que espaços definidos por parâmetros de entrada sejam limitados em sua abrangência e agrupados de tal forma que subespaços sejam formados. No desenvolvimento de *testbenches*, os geradores de estímulos aleatórios podem ser criados de forma a conter subespaços que se sobrepõem (resultando em estímulos redundantes) ou subespaços que contenham condições que não sejam de interesse (resultando em estímulos inválidos). É possível eliminar ou diminuir, os casos de teste redundantes e inválidos através da aplicação de metodologias de modificação do espaço de estímulos de entrada, e assim, diminuir o tempo requerido para completar a simulação de modelos HDL. No presente trabalho, é realizada uma análise da aplicação da técnica de organização do espaço de entrada através de domínios de parâmetros do IP, e uma metodologia é desenvolvida para tal, incluindo-se, aí, uma ferramenta de codificação automática de geradores de estímulos aleatórios em linguagem SyatemC: o *GET_PRG*. Resultados com a aplicação da metodologia é comparada a casos de aplicação de estímulos aleatórios gerados a partir de um espaço de estímulos de entrada sem modificações.

Como esperado, o número de casos de teste redundantes e inválidos aplicados aos *testbenches* foi sempre maior para o caso de estimulação aleatória a partir do espaço de estímulos de entrada completo com um tempo de execução mais longo.

ABSTRACT

One of the strongest restrictions that exist throughout ICs design flow is the need for shorter development cycles. This, along with the constant demand for more functionalities, has been the main cause for the appearance of the so-called System-on-Chip (SOC) architectures, consisting of systems that contain dozens of reusable hardware blocks (Intellectual Properties, or IPs). The increasing complexity makes it necessary to thoroughly verify such models in order to guarantee 100% functional applications.

Among the current verification techniques, functional verification has received important attention, since it represents an alternative that keeps HDL validation costs low throughout the circuit's design cycle. Functional verification is based in testbenches, and it works by exploring the whole (or relevant) model's functionality, applying test cases in a sequential fashion. This allows the testing of the model in all desired input sequences and combinations. There are different techniques concerning testbench design, being the random stimulation an important approach, by which a huge number of test cases can be automatically created.

In order to ease the stimuli application in circuit simulation, it is common to limit the range of the space defined by input parameters and to group such restricted parameters in sub-spaces. In testbench development, it may occur the creation of random stimuli generators containing overlapping sub-spaces (resulting in redundant stimuli) or sub-spaces containing conditions of no interest (resulting in invalid stimuli). It is possible to eliminate, or at least reduce redundant and invalid test cases by modifying the input stimuli space, thus, diminishing the time required to complete the HDL models simulation. In this work, the application of a technique aimed to organize the input stimuli space, by means of IP parameter domains, is analyzed. A verification methodology based on that is developed, including a tool for automatic coding of random stimuli generators using SystemC: *GET_PRG*. Results on applying such a methodology are compared to cases where test vectors from the complete verification space are generated.

As expected, the number of redundant test cases applied to the testbenches was always greater for the case of random stimulation on the whole (unreduced, unorganized) input stimuli space, with a larger testbench execution time.

SUMÁRIO

AGRADECIMENTOS	i
RESUMO	ii
ABSTRACT	iv
SUMÁRIO.....	v
LISTA DE FIGURAS	viii
LISTA DE TABELAS	x
LISTA DE ABREVIATURAS E SIGLAS	xi
1 INTRODUÇÃO.....	1
1.1 Aspectos de Verificação Funcional	3
1.2 Motivação e Justificativa	4
1.3 Objetivos.....	6
1.4 Organização do documento.....	7
2 VERIFICAÇÃO FUNCIONAL	9
2.1 Ambientes de Simulação.....	10
2.2 Cobertura Funcional	16
2.3 Ferramentas de Verificação	20
2.4 Plano de Verificação	23
3 OTIMIZAÇÃO DO ESPAÇO DE ESTÍMULOS DE SIMULAÇÃO.....	25
3.1 Domínios de Parâmetros de Verificação	25
3.2 Outros Trabalhos Relacionados	32
4 METODOLOGIA DE VERIFICAÇÃO COM PDGs.....	38

4.1	O Ambiente de Visualização Gráfica e Checagem de Parâmetros: <i>ParaGraph</i>	39
4.2	SystemC Verification Library, SCV	42
4.3	Geração do PDG e criação do Gerador de Estímulos em SystemC.....	44
4.4	A Ferramenta <i>GET_PRG</i> (<i>Generation Tool for Parameter-Domain-based Random Stimuli Generation</i>)	45
5	DOIS CASOS DE ESTUDO: UM ADAPTADOR <i>BLUETOOTH</i> BANDA BASE E UM DECODIFICADOR <i>MPEG-4</i>	49
5.1	Controlador da Camada Banda Base do Protocolo <i>Bluetooth</i>	49
5.2	Decodificador de Vídeo <i>MPEG-4</i>	57
5.3	Aplicação da Metodologia de Domínios de Parâmetros aos Modelos do Adaptador <i>Bluetooth</i> e o Decodificador <i>MPEG-4</i>	64
6	RESULTADOS	69
6.1	Metodologia de Simulação de Modelos e Comparação de Resultados	69
6.2	Caracterização do Espaço de Estímulos de Entrada	70
6.3	Redução Final do Tamanho do Espaço de Simulação	74
6.4	Redundância na Geração de Estímulos.....	75
6.5	Especificação dos Modelos de Cobertura Funcional	77
6.6	Análise Pós-Simulação da Cobertura Funcional.....	80
6.7	Análise Estatística de Vetores de Estímulos.....	87
7	CONCLUSÕES E CONSIDERAÇÕES FINAIS	92
	Referências bibliográficas	95
	Apêndice A.....	100
	Apêndice B	104

Apêndice C	107
Apêndice D	111
Apêndice E	113
Apêndice F	117

LISTA DE FIGURAS

Figura 1. Modelo geral de <i>testbench</i> .	11
Figura 2. Modelo de <i>testbench</i> reativo.	14
Figura 3a. <i>Testbench</i> para verificação do protocolo AMBA AHB (SU et al., 2006).	14
Figura 3b. <i>Testbench</i> para verificação do protocolo STBus.	15
Figura 4a. Verificação com reuso horizontal.	16
Figura 4b. Verificação com reuso vertical.	16
Figura 5. Motor de trabalho da maior parte das ferramentas de simulação.	21
Figura 6. Espaços de simulação sobrepostos.	27
Figura 7. Subespaços dos parâmetros de simulação que formam Domínios de Parâmetros.	28
Figura 8a. Espaço de simulação sem levar em conta dependências entre parâmetros.	31
Figura 8b. MDD representando o novo espaço de simulação reduzido através das dependências entre os parâmetros.	31
Figura 9. Seleção de testes importantes baseando-se em medições relativas de similaridade.	35
Figura 10. Comparação entre enfoques diferentes de cobertura.	37
Figura 11. Estrutura hierárquica de classes do <i>ParaGraph</i> .	40
Figura 12. Algoritmo do script Python para criação de geradores de estímulos SystemC.	45
Figura 13. Estrutura simplificada das diferentes camadas do padrão <i>Bluetooth</i> .	50
Figura 14. Formato do pacote da camada banda base <i>Bluetooth</i> .	50
Figura 15. Formato do Cabeçalho do Pacote.	51
Figura 16. Formato do cabeçalho da carga útil.	52
Figura 17. Operações realizadas sobre o cabeçalho do pacote.	54
Figura 18. Operações realizadas sobre a carga útil do pacote.	54
Figura 19. Diagrama hierárquico da implementação da camada banda base Bluetooth.	55
Figura 20. Composição Hierárquica dos Objetos de Vídeo <i>MPEG-4</i> .	58
Figura 21. Obtenção do vetor de movimento entre os blocos de duas imagens consecutivas.	62
Figura 22. Diagrama de Blocos da Implementação do Decodificador MPEG-4.	63
Figura 23. PDG obtido para submódulo SI.	72
Figura 24. Evolução global da cobertura do submódulo SI do decodificador <i>MPEG-4</i> .	82

Figura 25. Cobertura individual de itens dos parâmetros com dependências para uma simulação com geração tradicional.....	83
Figura 26. Cobertura individual de itens dos parâmetros sem dependências para uma simulação com geração tradicional.....	84
Figura 27. Cobertura individual de itens dos parâmetros com dependências para uma simulação com geração com domínios de parâmetros.	85
Figura 28. Cobertura individual de itens dos parâmetros sem dependências para uma simulação com geração com domínios de parâmetros.	86

LISTA DE TABELAS

Tabela 1. Tipos de Pacote <i>Bluetooth</i> , segundo código TIPO do cabeçalho, e tipo de enlace.	53
Tabela 2. Parâmetros de simulação para o STREAMPROC, e os seus respectivos possíveis valores.....	65
Tabela 3. Parâmetros de simulação para os módulos SI e CBP, e os seus respectivos valores.	67
Tabela 4. Relação do tamanho do espaço de parâmetros para os diferentes módulos.	74
Tabela 5. Estímulos inválidos e tempos de simulação para 200.000 casos de teste.....	75
Tabela 6. Modelo de cobertura do <i>Bluetooth</i>	78
Tabela 7. Modelo de cobertura do SI.	79
Tabela 8. Modelo de cobertura para o CBP.....	80
Tabela 9. Tempos de simulação sob um modelo de cobertura funcional por itens.	81

LISTA DE ABREVIATURAS E SIGLAS

ACL	<i>Asynchronous Connection-Less Channel</i>
AHB	<i>Advanced High-performance Bus</i>
AMBA	<i>Advanced Microcontroller Bus Architecture</i>
ATPG	<i>Automatic Test Pattern Generation</i>
CI	<i>Circuito Integrado</i>
CRC	<i>Cyclic Redundancy Check</i>
DCT	<i>Discrete Cosine Transform</i>
DH	<i>Data High Rate</i>
DM	<i>Data Medium Rate</i>
DUV	<i>Design Under Verification</i>
FEC	<i>Forward Error Correction</i>
FPGA	<i>Field-Programmable Gate-Array</i>
FHS	<i>Frequency Hopping Synchronization</i>
FSM	<i>Finite State Machine</i>
HEC	<i>Header Error Check</i>
HDL	<i>Hardware Description Language</i>
HVL	<i>Hardware Verification Language</i>
IP	<i>Intellectual Property</i>
L2CAP	<i>Logic Link Control and Adaptation Protocol</i>
LAP	<i>Lower Address Part</i>
LMP	<i>Link Manager Protocol</i>
MDD	<i>Multiple Decision Diagram</i>
MPEG	<i>Moving Picture Experts Group</i>
PDG	<i>Parameter Domain Graph</i>

RTL	<i>Register Transfer Level</i>
SCO	<i>Synchronous Connection-Less Channel</i>
SOC	<i>System-on-Chip</i>
UAP	<i>Upper Address Part</i>
VHDL	<i>VHSIC Hardware Language</i>
VHSIC	<i>Very High Speed Integrated Circuit</i>
VLC	<i>Variable Length Code</i>

1 INTRODUÇÃO

O projeto de circuitos integrados (CIs) tem atingido um alto nível de complexidade, decorrente dos avanços do processamento tecnológico e das exigências e necessidades do mercado tecnológico recente. Evidência disso é o rápido e contínuo crescimento do número de elementos e funções presentes nos circuitos eletrônicos que, por sua vez, devem diminuir, ou pelo menos manter, as dimensões físicas das aplicações associadas (sejam independentes ou embutidas).

Uma das maiores restrições que existe atualmente no fluxo de projeto de CIs é a necessidade de um ciclo menor no seu desenvolvimento, requerendo maior eficiência ao longo de todo o projeto. Isto, junto à constante demanda por maior número de funcionalidades, deu origem às denominadas arquiteturas SOC (*System-On-Chip*), que resultam em sistemas até dúzias de blocos de hardware reusável (*IPs* ou *Intellectual Properties*). Devido às grandes dimensões dos sistemas atuais, é muito provável encontrar no projeto de blocos IP, erros ou *bugs* originados na passagem de uma dada especificação inicial para seus correspondentes modelos em linguagem de descrição de hardware (*hardware description language*, HDL). Considerando-se os padrões atuais de qualidade total na fabricação, os erros da etapa de desenvolvimento tornam-se inadmissíveis, por representarem custos adicionais de projeto e iterações desnecessárias no ciclo de produção.

É bem conhecido o fato de as tarefas de depuração do projeto em HDL consumirem entre 60% e 70% do esforço total do projeto (JERINIĆ, 2005). Esta situação tem despertado nos meios acadêmico e industrial uma preocupação generalizada em melhorar a eficiência da verificação de SOC.

Uma particularidade considerada positiva nos IPs é a possibilidade de serem adaptados a um maior número de diferentes aplicações graças ao seu (em geral) abrangente conjunto de parâmetros, isto é, a quantidade e o tipo de dados que recebem ou entregam. De um lado, um amplo conjunto de parâmetros aumenta a possível faixa de utilização do IP, facilitando sua aplicação em múltiplos produtos, e amortizando os custos de desenvolvimento e aquisição. Por outro lado, torna-se mais difícil haver a segurança de que o IP se comporta corretamente ante as configurações requeridas pelos projetistas. Cada parâmetro adicional aumenta o espaço de verificação e, por conseguinte, o custo de validação do sistema eleva-se consideravelmente (JERINIĆ, 2005).

A verificação dos blocos de propriedade intelectual pode ser realizada através de técnicas formais como checagem de modelos, prova de teoremas, e checagem de equivalência. A primeira procura demonstrar que as propriedades definidas para o sistema nunca serão infringidas, para qualquer possível seqüência de entrada. A segunda fundamenta-se em demonstrar que teoremas referentes a um determinado sistema são verdadeiros através de um motor de prova. Por último, a checagem de equivalência compara dois modelos para estabelecer se eles são (ou não) logicamente equivalentes, independentemente do seu nível de abstração, e de componentes físicos como *latches* e registradores (YUAN; PIXLEY; AZIZ, 2006). No entanto, devido ao grande tamanho dos blocos de propriedade intelectual da atualidade, os métodos formais têm tido aplicação limitada, motivo pelo qual o método de verificação funcional tem-se imposto, sendo este fato evidenciado pela grande quantidade de ferramentas existentes para este fim, como o Synopsys™ Vera, o Cadence™ TestBuilder, o Verisity™ Specman, etc. (ROMERO, 2005)

A verificação funcional (ou também, verificação dinâmica) é baseada em simulação do projeto, tendo como objetivo explorar ao máximo a funcionalidade do modelo, aplicando-se casos de teste, um após outro. Isto permite examinar o modelo em todas as seqüências e combinações de entradas consideradas relevantes pela equipe de verificação do projeto (YUAN; PIXLEY; AZIZ, 2006).

1.1 Aspectos de Verificação Funcional

Uma vez que se tem um modelo ou protótipo de hardware pronto, o engenheiro ou a equipe de verificação deve estabelecer e estruturar o plano de verificação deste modelo. O plano deve conter todos os passos, recursos e procedimentos (por exemplo, metodologia de verificação, equipamento e pessoas necessárias, modelos a serem verificados, propostas do modelo de cobertura, etc.) que se pretende adotar, antes, durante, e após, a verificação (BERGERON, 2003). Dentre as tarefas a serem incluídas no plano de verificação, destacam-se, pela sua relevância, a geração dos estímulos e a avaliação da cobertura funcional.

A eficiência da verificação funcional, que é a metodologia a ser seguida no presente trabalho, é passível de ser drasticamente afetada pela estratégia de geração dos vetores de teste proposta. Por exemplo, um dispositivo que tem N entradas e M flip-flops, requer $(2^N)^M$ estímulos para que seja testado na sua integridade. Se o dispositivo tiver 10 entradas, e 100 flip-flops, ele requererá 2^{1000} vetores de teste; a uma taxa de 1000 vetores por segundo, a simulação duraria aproximadamente 3.4×10^{308} anos. Fica clara a necessidade de se estimular o modelo de forma apropriada para não se criar gargalos ou obstáculos que impeçam a satisfação dos itens de cobertura previamente definidos (PIZIALI, 2004). Diferentes estratégias de aplicação de estímulos têm sido utilizadas, muitas vezes dentro do mesmo projeto, podendo ser classificadas em estimulação direta, de casos reais, de casos extremos e aleatórios.

A cobertura é um critério que permite avaliar o nível de abrangência atingido pela verificação em curso, ou seja, é uma métrica que fornece um guia para dirigir a verificação funcional. A cobertura completa (100%) indica que todas as metas estabelecidas para a verificação são alcançadas. A cobertura é um aspecto de difícil generalização, pois define o que é relevante de ser observado e o momento em que a verificação de um modelo pode ser considerada concluída. Entretanto, muitos especialistas da área de verificação a tem como o aspecto mais relevante de ser especificado detalhadamente no plano de verificação, já que é o elemento que eventualmente permitirá conhecer objetivamente o nível de qualidade do IP e integridade da verificação funcional.

A cobertura pode ser do tipo estrutural, concentrando-se em observar quais aspectos do código da implementação do projeto sob verificação (*design under verification*, DUV), ou seja, da sua representação em código HDL, sejam cumpridos. A cobertura estrutural, ou de

código, pode servir para medir quais e/ou quantas vezes os sinais e latches do DUV mudam de valor lógico (*toggle coverage*), linhas de código sequencial são executadas (*statement coverage*), expressões condicionais são alcançadas (*branch coverage*), caminhos de código são percorridos (*path coverage*), ou finalmente, estados ou transições da FSM (máquina de estados finitos) que representa o DUV são estimulados (*FSM coverage*) (YUAN; PIXLEY; AZIZ, 2006).

A grande debilidade da cobertura estrutural está no fato de que estímulos aplicados a sinais individuais, ou linhas de código HDL, não permitem desenvolver uma noção da significância funcional da verificação executada (WILE; GOSS; ROESNER, 2005). Por outro lado, a cobertura funcional focaliza a criação de um ambiente de simulação no qual estímulos são gerados, objetivando testar funções ou conjuntos de funções específicas do DUV, estipuladas na especificação inicial. Entre os esquemas de cobertura funcional existentes, destacam-se os seguintes: cobertura funcional por itens, cobertura funcional de combinações e cobertura funcional de transições. A cobertura funcional por itens visa à checagem da presença (e quantidade) de valores escalares dos parâmetros de estimulação e de observação considerados relevantes para os objetivos da verificação, enquanto a cobertura de combinações visa a checagem da ocorrência de combinações de valores; finalmente, a cobertura de transições visa a checagem da ocorrência de seqüências destes valores escalares (ROMERO; STRUM; CHAU, 2005). A tarefa do projetista é, então, definir um modelo de cobertura adequado às características do projeto que se deseja verificar, para reduzir ao mínimo a probabilidade de não detecção de erros no código HDL.

Dentre os possíveis modelos de cobertura funcional, os modelos de itens de cobertura têm recebido bastante atenção e popularidade recentemente (WILE; GOSS; ROESNER, 2005). A importância e a vantagem de se desenvolver uma técnica de verificação funcional focalizada em itens de cobertura reside na possibilidade de estabelecer um espaço bem definido de verificação, ou seja, conhecer com exatidão as características, funções, e elementos do DUV que se espera verificar, assim como, o nível de completude e qualidade de tal verificação.

1.2 Motivação e Justificativa

Diferentes técnicas têm sido documentadas no que se refere à construção dos ambientes de simulação, ou *testbenches*. Análises realizadas sobre resultados colhidos de *testbenches* têm

mostrado que a progressão da cobertura funcional no tempo, num processo de verificação, é superior para o caso de aplicação de estímulos aleatórios comparados ao de estímulos dirigidos (BERGERON, 2003). Além disso, a estimulação aleatória tem-se mostrado muito vantajosa para a criação automática de um grande número de casos de teste, abrangendo itens diversos da funcionalidade do DUV. Por outro lado, do ponto de vista da cobertura, ela é uma grande fonte de casos de teste irrelevantes, à medida que a execução do *testbench* progride. Entende-se, aqui, como estímulos irrelevantes todos aqueles casos de teste que não representam contribuição nenhuma para o avanço da cobertura estabelecida, porque correspondem a condições já suficientemente exploradas. Técnicas eficientes para redução do tempo de execução de *testbenches* são possíveis como é o caso do modelo de *testbench* reativo por filtragem de estímulos irrelevantes¹ (ROMERO, 2005) ou por direcionamento através da cobertura (VERMA; HARRIS; RAMINENI, 2007) (REGIMBAL et al., 2003); tais técnicas reduzem significativamente o tempo de execução dos *testbenches* aplicados em diferentes blocos de hardware, mas, dado que o gerador de estímulos neste caso ainda é essencialmente aleatório, alguns problemas perduram.

Para facilitar a simulação, é comum agrupar-se conjuntos de casos de teste, com o espaço de parâmetros dividido em subespaços, limitando-se a abrangência dos valores de alguns desses parâmetros. Como a construção de um gerador de estímulos aleatórios é feita, em geral, manualmente, a divisão em subespaços pode trazer ineficiências na forma de casos de testes redundantes (subespaços sobrepostos) ou inválidos (subespaços fora da especificação). Tais fenômenos atrasam o progresso da cobertura e influem de forma indesejável no desempenho global da verificação.

É possível determinar integralmente o espaço desejado de verificação funcional, para os parâmetros de entrada, através de um modelo de cobertura predefinido (*pré-simulação*) como apresentado em (JERINIĆ, 2005), onde estímulos redundantes e inválidos, de uma maneira sistemática, são ignorados e eliminados do conjunto de possíveis valores que os parâmetros de simulação podem adquirir. Naquele trabalho, o conjunto completo de parâmetros é representado por um tipo especial de grafo de decisão denominado PDG (*Parameter Domain Graph*), onde os ramos determinam domínios de parâmetros, de forma que, qualquer par de

¹ Na referência (ROMERO, 2005), o termo utilizado é redundante ao invés de irrelevante. Neste texto o termo redundante é reservado para o caso de subespaços sobrepostos como será visto mais à frente.

ramos do grafo não pode conter condições de parâmetros idênticas e todo ramo constitui-se de condições válidas. Esta técnica tem se mostrado capaz de aumentar significativamente a velocidade da verificação ao serem conhecidos, de antemão, todos os casos de teste, sem se depender das respostas observadas à saída do DUV.

Apesar de o trabalho em (JERINIĆ, 2005) ter como alvo a geração direta exaustiva de estímulos, portanto, apenas para casos de circuitos pouco complexos, a técnica baseada em domínios de parâmetros pode ser aplicada também para circuitos de alta complexidade na estratégia de estimulação aleatória.

1.3 Objetivos

O principal objetivo do presente trabalho é analisar a influência da aplicação do formalismo por grafos de domínios de parâmetros, introduzidos e utilizados em (JERINIĆ, 2005) e (JERINIĆ; MÜLLER, 2004), sobre o desempenho da verificação funcional, fazendo-se uma avaliação comparativa com as metodologias tradicionais de verificação, em termos de eficiência. Assim, deseja-se aumentar a confiança no processo de verificação, utilizando-se critérios objetivos para o estabelecimento da integralidade dos testes, facilitando a geração de casos de teste aleatórios que maximizem a cobertura funcional.

Este projeto prevê o atendimento dos seguintes objetivos específicos:

1. A partir das considerações introduzidas em (JERINIĆ, 2005) e (JERINIĆ; MÜLLER, 2004), estabelecer uma metodologia conceitual otimizada, e estruturada, para sistematizar o processo de verificação baseada no formalismo de domínios de parâmetros.
2. Desenvolver uma ferramenta que, a partir da descrição do espaço de verificação definido por PDGs, crie automaticamente os geradores de estímulos para casos de valores aleatórios e extremos, utilizando a linguagem de descrição de sistemas SystemC. Esta ferramenta, implementada em forma de script na linguagem Python, recebeu o nome *GET_PRG*.
3. Apresentar resultados de aplicação de estímulos aleatórios, e estímulos orientados por domínios de parâmetros, na verificação de sistemas realistas (de grandes dimensões), como o adaptador de rede *Bluetooth*, e o decodificador de vídeo *MPEG-4*.

4. Agregar toda a metodologia de domínios de parâmetros à mesma técnica tradicional apresentada em (ROMERO; STRUM; CHAU, 2005) e compará-la ao método de estímulos aleatórios simples.

1.4 Organização do documento

A seguir, a forma como este documento está dividido é apresentada.

O segundo capítulo apresenta um estudo sobre a verificação funcional, mostrando aspectos como estratégias, modelos de *testbench*, e reuso de componentes de verificação. Também é feita uma breve apresentação de conceitos de cobertura funcional.

O terceiro capítulo apresenta um estudo sobre estratégias destinadas a acelerar as tarefas da verificação funcional através de uma análise inicial e da otimização do espaço de estímulos de entrada do DUV. Vários trabalhos sobre o assunto são descritos neste capítulo, porém a ênfase é para a metodologia dos domínios de parâmetros de verificação, base conceitual do presente trabalho.

O quarto capítulo mostra todos os aspectos técnicos específicos da metodologia de verificação com domínios de parâmetros. O capítulo inclui a apresentação de recursos de terceiros necessários à metodologia, como a ferramenta e biblioteca *ParaGraph*, e a biblioteca de verificação da linguagem SystemC; também são apresentados os detalhes da adaptação da metodologia à linguagem SystemC através da ferramenta *GET_PRG*, desenvolvida no âmbito deste trabalho, para a codificação automática de geradores de estímulos.

O quinto capítulo apresenta o estudo realizado sobre dois núcleos de hardware que serviram como caso de estudo: o adaptador banda base *Bluetooth* e o decodificador de vídeo *MPEG-4*. São mostrados os elementos do protocolo *Bluetooth* e do padrão *MPEG-4* encontrados nas respectivas especificações, além das características das respectivas implementações em RTL. Também, para os módulos e submódulos estudados, os parâmetros utilizados na aplicação da metodologia são enumerados e explicados.

O sexto capítulo mostra os resultados da aplicação da metodologia proposta de verificação às implementações *Bluetooth* e *MPEG-4*, e a comparação, quantitativa e qualitativa, com os

resultados de uma verificação tradicional.

Finalmente, o sétimo capítulo apresenta as considerações finais e são indicadas as propostas para trabalhos futuros de pesquisa nesta área.

2 VERIFICAÇÃO FUNCIONAL

O projeto de sistemas eletrônicos integrados (*Systems-on-Chip*, SOCs) está baseado na escolha de plataformas preestabelecidas, conhecidas como blocos de propriedade intelectual, ou simplesmente, IPs. Geralmente, esses blocos se caracterizam por serem altamente integráveis às aplicações às que se destinam, uma vez que, em regra, eles são projetados sob um alto nível de parametrização, que se traduz em adaptabilidade, versatilidade, e mais importante, reusabilidade. O reuso de IPs tem aparecido como resposta à exigência por aumento da eficiência, e redução de tempo de desenvolvimento; entretanto, a sua aceitação requer um exaustivo processo de verificação do IP para garantir a sua qualidade (em outros termos, a sua equivalência à especificação).

A tarefa de verificação é um dos grandes gargalos no projeto de circuitos integrados, consumindo grande quantidade de recursos humanos e materiais. Desta forma, inúmeras abordagens diferentes foram propostas para aumentar a eficiência da verificação.

Uma das estratégias propostas para realizar a verificação consiste na adoção de métodos de verificação formal, que utilizam ferramentas de análise exaustiva de propriedades de modelos de interesse; um método bastante conhecido é a de checagem de equivalência booleana que analisa o modelo no nível de porta lógica ou o de transferência entre registradores, RT. A verificação formal é a abordagem mais confiável por conseguir determinar com certeza a exatidão das propriedades do modelo sendo verificado, mas é ineficiente no tratamento de sistemas complexos, de grandes dimensões, com um grande número de estados (PIXLEY et al., 2003). Alternativamente, é possível aplicar estímulos (geralmente aleatórios e dirigidos por cobertura) a um modelo em software do circuito através de um ambiente de simulação

para testar as características ou funções consideradas essenciais. Esta técnica é chamada de verificação funcional, e sua maior vantagem, até hoje, em comparação à verificação formal, encontra-se no fato de requerer muito menos tempo e recursos. Na verificação funcional, o modelo do projeto a ser verificado (*design under verification*, DUV) em linguagem de descrição de hardware (*hardware description language*, HDL) é executado dentro do ambiente de simulação que permite avaliar a reação do modelo a estímulos específicos. O engenheiro de verificação deve, então, checar os resultados registrados pelo ambiente de simulação, e se for o caso, assinalar a ocorrência de comportamentos errôneos (WILE; GOSS; ROESNER, 2005).

Finalmente, existe a possibilidade de aplicar estímulos a um modelo físico, carregado em uma plataforma de hardware reconfigurável, comumente um arranjo de portas programáveis em campo (*field-programmable gate-arrays*, FPGAs), o que permite acelerar enormemente a verificação e testar o modelo implementado fisicamente. Porém, esta última metodologia, que é chamada de emulação ou aceleração de hardware, costuma ser extremamente custosa e requer alta capacidade de processamento e velocidade no armazenamento de dados.

2.1 Ambientes de Simulação

O *testbench*, ou ambiente de simulação, modela o ambiente real de aplicação e utilização do DUV. Assim, seria desejável que tal ambiente suportasse todas as ações que possam acontecer com o circuito a ser verificado. Dada a especificidade de cada projeto, e a necessidade de adaptar o ambiente de simulação aos seus parâmetros e características, não é possível uma abordagem única para tratar o problema da verificação funcional. Pode-se dizer que há, praticamente, uma metodologia diferente de verificação para cada projeto de circuitos integrados que tenha sido realizado até o presente. O *testbench* mais básico consiste de: o circuito ou lógica que está sendo verificada, o(s) gerador(es) de estímulos, e os componentes de determinação de erro (WILE; GOSS; ROESNER, 2005). Alguns ambientes podem incluir adicionalmente elementos de registro de eventos e/ou dados (*scoreboarding*).

Um dos modelos de *testbench* mais genéricos e utilizados é o proposto em (BERGERON, 2003), ilustrado na Figura 1. Este modelo, como exposto na ilustração, apresenta dois tipos de sinais: os canais abstratos, que representam transações ou valores dos parâmetros de simulação em alto nível, independentes do tempo; e sinais de tempo, que transportam

informações de baixo nível correspondentes aos sinais que o DUV utilizaria na sua operação real.

Figura 1. Modelo geral de *testbench*.

O ponto de partida deste modelo é o gerador de estímulos. Neste bloco, estímulos são gerados no nível de transações, ou seja, representações abstratas das entradas ou saídas do DUV. Há basicamente duas funções no bloco de amarração: primeiramente, o *Driver*, onde as transações provenientes do gerador de estímulos são traduzidas em sinais de tempo, para que possam ser inseridas no DUV (o objeto do teste); depois, o *Monitor*, onde as reações ou respostas do DUV em sinais de tempo são observadas e transcritas de novo em termos de transações para fins de comparação no bloco de auto-checagem. A funcionalidade é testada através do módulo de auto-checagem, o qual pode conter, por exemplo, um modelo de referência, normalmente descrito em linguagens de alto nível como o C++, MatLab, SystemC, SystemVerilog, etc., cujas respostas aos mesmos sinais de estímulos inseridos no *Driver*, são comparadas aos resultados do Monitor (ROMERO, 2005).

2.1.1 Estratégias de Estimulação

Os estímulos a serem gerados dentro do bloco que efetua as funções de teste (Figura 1) podem se enquadrar em quatro categorias:

- estimulação direta: esta é uma forma de verificar características específicas do projeto através de estímulos programados para tal. Muitas vezes o conjunto de estímulos é recomendado na especificação do IP. A maior desvantagem desta alternativa é a sua falta de escalabilidade, tornando-se incontrolável para quantidades elevadas de casos de teste (BERGERON, 2003).

- estimulação por casos reais: os casos reais de teste são aqueles compostos, unicamente, por estímulos que representam dados reais de operação do sistema. A relevância deste tipo de estimulação reside na necessidade de se monitorar, e verificar, o funcionamento do modelo do sistema, sob condições reais de operação. Exemplos incluem cadeias de bits correspondentes a áudio, imagens, e vídeo; dados de autenticidade, segurança, e qualidade de informação; seqüências de dados de controle, etc.
- estimulação de casos extremos: existem vários estados dos modelos que devem ser verificados com especial atenção, uma vez que, geralmente, são estados ativados por valores extremos dos parâmetros de simulação, em situações específicas ou pouco usuais, porém possíveis. Outros métodos de estimulação (aleatórios, diretos, reais) podem fornecer certo nível de eficiência, escalabilidade, e reusabilidade, mas dificilmente conseguem envolver o nível de cobertura necessário para atingir erros relacionados a casos extremos dos valores dos estímulos de entrada (FUJITA; GHOSH; MUKUL, 2008). A estimulação de casos extremos pode ser considerada um tipo especial de estimulação direta, na qual tenta-se descobrir erros de projeto que possivelmente seriam detectados somente após um grande número de ciclos inúteis de simulação.
- estimulação aleatória: os estímulos gerados aleatoriamente são a técnica mais comum de automatização dos casos de teste, necessária para realizar a verificação dos sistemas digitais atuais, para os quais os espaços de estímulos de entrada são de grande magnitude. Os ambientes de simulação que utilizam estímulos aleatórios, com as restrições apropriadas para não gerar estados ilegais do sistema, são efetivos ao gerar uma grande quantidade de estímulos, que ajudam ao projetista descobrir erros em condições que não se consideraria inicialmente. As ferramentas existentes para a construção de *testbenches* contêm componentes de geração de estímulos que usam geradores de estímulos pseudo-aleatórios, e tabelas de probabilidade para gerar decisões sobre os estímulos de entrada (WILE; GOSS; ROESNER, 2005). Tais tabelas de probabilidade devem ser configuradas previamente à simulação, para influenciar as decisões do componente em relação aos valores a gerar, e a frequência de geração desses valores.

Em (BERGERON, 2003) recomenda-se que os *testbenches* devam ter auto-checagem e operar utilizando-se transações, enquanto os estímulos devem ser gerados de forma aleatória, com

restrições (limites de valores) definidas. A estimulação aleatória é necessária para cobrir espaços de estímulos que de outra maneira seriam intratáveis devido às suas dimensões, enquanto as restrições, como mencionado anteriormente, são estabelecidas para enquadrar os estímulos dentro dos limites dos valores de operação correta do dispositivo. Contudo, também é possível restringir, ou dirigir, os estímulos gerados, para otimizar os critérios de cobertura (Seção 2.2) definidos no plano de verificação do projeto. Como estratégia geral de verificação, a combinação da estimulação aleatória com a estimulação direta, ou dirigida (para estimular áreas específicas de valores dos parâmetros de simulação), produz um aumento da eficiência da verificação significativo, ao aumentar a taxa de ocorrência daqueles estímulos que exercem funções do sistema que são de especial interesse.

2.1.2 Tipos de *Testbench*

Existem duas categorias principais em que os ambientes de simulação podem ser classificados: ambientes de malha aberta, onde as respostas observadas à saída do DUV são utilizadas unicamente para fins de checagem de erros de projeto, e, ambientes de malha fechada, também denominados de reativos, onde a geração de estímulos é influenciada ou redirigida a partir da informação fornecida pelas respostas do DUV a estímulos anteriores.

Dentre os *testbenches* de malha fechada, existe uma tendência metodológica que tem obtido grande popularidade e aceitação, consistente na aplicação de estímulos filtrados a partir da observação dos casos de teste previamente simulados. Em (ROMERO, 2005), por exemplo, é desenvolvido um modelo de *testbench* reativo na linguagem SystemC, que permite diminuir a redundância dos estímulos aleatórios gerados e que é apresentado na Figura 2. Embora exista uma computação extra relacionada à tomada de decisão de aplicação ou exclusão de um estímulo no bloco de filtragem, a eliminação dos estímulos redundantes acaba melhorando consideravelmente a eficiência de simulação.

Figura 2. Modelo de *testbench* reativo.

Também, em (SU et al., 2006) e (LAHBIB et al., 2006) são desenvolvidos modelos de *testbench* reativos para o estudo de casos específicos de interfaces de comunicação. Em (SU et al., 2006), é apresentada a caracterização de um ambiente de simulação que permite verificar o funcionamento do protocolo AMBA AHB (*Advanced High-performance Bus*, protocolo definido na especificação da interface *Advanced Microcontroller Bus Architecture*, AMBA), mediante relatórios de cobertura que são iniciados a partir de uma descrição de transações baseadas em máquinas de estados finitos. Em (LAHBIB et al., 2006), foca-se a geração de estímulos aleatórios na linguagem SystemC para a simulação de uma interface do protocolo *STBus*, usando-se métricas de cobertura funcional para dirigir os casos de teste. As Figura 3a e Figura 3b ilustram tais abordagens de maneira geral.

Figura 3a. *Testbench* para verificação do protocolo AMBA AHB

(SU et al., 2006).

Figura 3b. Testbench para verificação do protocolo STBus
(LAHBIB et al., 2006).

2.1.3 Reuso de Componentes de Verificação

Um aspecto vantajoso no projeto de *testbenches* é a possibilidade de explorar a modularidade. Da mesma forma que o reuso de blocos de hardware, ou IPs, melhora o desempenho e tempo de produção de sistemas sobre silício, o reuso de alguns componentes, ou módulos de verificação, aumenta o desempenho global da verificação funcional. O resultado, então, é um melhor *time-to-market* (tempo de colocação do produto no mercado), assim como redução dos recursos requeridos (BERGERON, 2003). Existem dois tipos de arquiteturas de reuso de componentes de verificação que podem ser implementadas de forma eficiente, dependendo das características do projeto, e os requerimentos do plano de verificação (Seção 2.4):

- arquiteturas horizontais: este tipo de arquitetura se apresenta tanto no reuso de blocos de hardware, quanto no reuso de componentes de verificação num mesmo nível hierárquico do sistema digital. A Figura 4a apresenta um exemplo de implementação de uma arquitetura de verificação com reuso horizontal (WILE; GOSS; ROESNER, 2005). À esquerda, são mostrados três blocos de hardware, A, B, e C, que possuem interfaces de comunicação idênticas. Aproveitando-se da similaridade, a verificação desses blocos pode ser feita utilizando-se o mesmo componente de estimulação nos três casos (Figura 4a, blocos em cinza).
- arquiteturas verticais: reuso de componentes de verificação através de dois ou mais níveis hierárquicos do sistema. Pode-se dizer que esta arquitetura de reuso é exclusiva da verificação, já que blocos de hardware, ou IPs, dificilmente poderão ser adaptados em vários níveis hierárquicos do sistema. Como ilustrado na Figura 4b, os

componentes de estimulação e checagem, utilizados para verificar o bloco de hardware A, podem servir para verificar níveis hierárquicos superiores de hardware que utilizem este mesmo bloco (WILE; GOSS; ROESNER, 2005).

Figura 4a. Verificação com reuso horizontal.

Figura 4b. Verificação com reuso vertical.

Como poderá ser observado em capítulos posteriores, as duas formas de reaproveitamento de módulos de verificação preexistentes foram utilizadas neste trabalho.

2.2 Cobertura Funcional

Devido à grande quantidade de estados que compõem os atuais projetos de sistemas digitais, se faz necessário definir, previamente à simulação, uma estratégia de medição do conjunto de características do projeto que se deseja ou que se precisa verificar.

Os blocos e sub-blocos de hardware dos circuitos integrados atuais apresentam arquiteturas com grandes espaços de memória, interfaces de comunicação de alta velocidade, canais de processamento de dados paralelos, e entre outras características, uma grande quantidade de barramentos; devido a isto, a tarefa de validação desses módulos de hardware deve ser eficiente e planejada com ajuda de métricas que forneçam uma visão clara e objetiva das propriedades que são verdadeiramente relevantes, sem desperdiçar tempo e recursos em processos de verificação desnecessários. A cobertura funcional é, então, a métrica utilizada para medir o progresso da verificação, baseando-se na estimativa da funcionalidade coberta do projeto sob verificação, na simulação do modelo em HDL (WILE; GOSS; ROESNER, 2005). Quanto mais alta a cobertura obtida, mais aspectos relevantes do projeto terão sido monitorados.

Os geradores de estímulos aleatórios são, usualmente, a melhor opção para cobrir rapidamente a maior parte dos estados do circuito simulado. Por exemplo, um gerador completamente aleatório (sem restrição nenhuma além das condições necessárias para não criar valores ilegais de excitação), rodando durante tempo suficiente, conseguirá cobrir aproximadamente 75% ou 80% do espaço de funcionalidade total do circuito (HEKMATMPOUR; COULTER, 2003). No entanto, para verificar também o espaço restante (mormente composto por casos extremos e cenários complexos), a estimulação aleatória torna-se ineficiente, sendo indispensável combiná-la com diretivas de estimulação coerentes com aquelas condições especiais de cobertura funcional.

Complementar à cobertura funcional, é possível se definir um modelo de cobertura estrutural, ou de código, para realizar a verificação. Contudo, uma alta cobertura estrutural não necessariamente está correlacionada a uma alta cobertura funcional. Enquanto a cobertura estrutural está relacionada ao registro da execução mecânica do código HDL, a cobertura funcional refere-se mais à intenção ou propósito das funções implementadas. Por exemplo, é possível que a decodificação de uma instrução específica de uma CPU não tenha sido corretamente exercitada, mesmo se as linhas de código que a compõem tiverem 100% de cobertura (BERGERON, 2003). Em termos gerais, as duas abordagens deveriam ser complementares de maneira a cobrir, com ajuda de um modelo de cobertura estrutural, “buracos” que não possam ser preenchidos no modelo de cobertura funcional ou vice-versa.

A cobertura funcional pode ser definida como uma “ferramenta” que ajuda no monitoramento das funções do DUV que tenham sido exercitadas ao longo da simulação, evitando-se assim que escapem erros de projeto ocultos, que estariam associados a valores dos parâmetros de simulação ainda não testados. Este é um dos aspectos mais sutis da verificação funcional, pois dada a complexidade dos projetos digitais atuais, é difícil de se estabelecer o ponto até o qual é necessário exercitar o DUV para garantir que um determinado tipo ou quantidade de erros de projeto tenham sido descartados, ou que certo nível de cobertura funcional tenha sido atingido. Existem, no entanto, várias formas de organizar a cobertura funcional para se obter núcleos de hardware suficientemente depurados (isto é, que satisfaçam a sua especificação inicial com um nível aceitável de confiabilidade).

2.2.1 Cobertura Funcional por Itens

Uma das formas mais usuais de elaborar modelos de cobertura funcional é através de itens de cobertura. A cobertura funcional por itens consiste em um conjunto de eventos relevantes e de especial interesse que se pretende observar à entrada, dentro, ou à saída do DUV. Tais eventos devem ser definidos em função dos valores, ou conjuntos de valores, que os parâmetros de simulação do DUV podem adquirir, e do número de itens correspondente a cada um dos eventos definidos, ou seja, a quantidade de vezes que o evento deve ocorrer para se considerar satisfeito. Parâmetros podem ser variáveis do sistema ou os próprios parâmetros de configuração do sistema ou bloco. Exemplos de tais parâmetros podem ser: endereços de memória, comprimento de pacotes de dados, variáveis de dados ou controle, níveis de interrupções, estado de terminação de transações, nível de buffers, padrões de requisição de barramentos, etc.

Através da experiência do projetista de verificação, deve-se levantar e analisar os eventos de cobertura quanto à sua relevância e distribuição de valores. Se, por exemplo, for necessário medir a cobertura funcional de um endereço de 32 bits, o número total de eventos de cobertura corresponderia a aproximadamente 4 bilhões de valores. Neste caso, o melhor seria estabelecer eventos de cobertura funcional representados por conjuntos de valores de verdadeira importância. Dependendo da arquitetura do endereço em questão, por exemplo, os seus valores possíveis poderiam ser agrupados em quatro conjuntos que constituiriam os eventos relevantes de cobertura funcional, como na forma a seguir (YUAN; PIXLEY; AZIZ, 2006):

Evento 1	————→	Endereços que vão de 00000001 até 7FFFFFFF
Evento 2	————→	Endereços que vão de 80000000 até 8FFFFFFF
Evento 3	————→	Endereços que vão de A0000000 até CFFFFFFF
Evento 4	————→	Endereços que vão de D0000000 até FFFFFFFF

Existe, entretanto, um problema relacionado ao tamanho numérico de cada um desses eventos de cobertura, já que a faixa de valores de cada evento é ainda ampla demais para se estabelecer níveis exaustivos de cobertura. A forma de resolver este problema é determinar um número suficientemente representativo de itens a serem monitorados em cada faixa. No caso anterior, por exemplo, poder-se-iam definir 100.000 itens para cada um dos quatro eventos de cobertura funcional.

2.2.2 Cobertura Funcional por Combinações

Existem, também, estratégias de cobertura funcional por combinações, ou cobertura cruzada, fundamentada em modelos de cobertura construídos através de métodos de seleção de casos de teste, que são identificados por combinação de valores dos diferentes parâmetros do DUV. No trabalho apresentado em (GRINDAL; OFFUTT; ANDLER, 2004), aparecem classificadas a grande maioria de metodologias para se criar modelos de cobertura por combinações. As mais destacadas são as metodologias determinísticas que se dividem em instantâneas (OA, *orthogonal arrays* e CA, *covering arrays*), iterativas (EC, *each choice*, PPW, *partly pair-wise*, BC *base choice*, AC, *all combinations*, e AR, *anti-random*), e baseadas em parâmetros (IPO, *in parameter order*). Uma discussão mais detalhada da cobertura por combinações vai além do escopo do presente trabalho, ainda que se recomende a sua consulta em outras fontes bibliográficas.

Comumente, critérios de cobertura cruzada são necessários quando houver dependências entre dois ou mais parâmetros de simulação, à entrada, no interior, ou à saída do projeto sob verificação. O método mais básico para se definir eventos de cobertura cruzada, consiste no uso dos operadores lógicos elementares AND e OR. Isto, com o objetivo de medir unicamente a coincidência de valores específicos de vários parâmetros de simulação.

Voltando ao exemplo da Seção 2.2.1, onde o espaço de valores de um parâmetro foi dividido em quatro eventos de cobertura, é possível que cada um daqueles espaços de valores do

endereço de 32 bits estivesse relacionado a valores específicos de outro(s) parâmetro(s). Considerando que este parâmetro suposto recebe o nome “número_endereço”, e que pode tomar os valores 0, 1, 2, ou 3, seria possível definir os seguintes eventos de cobertura cruzada:

(número_endereço = 0) AND (endereço >= 00000001) AND (endereço <= 7FFFFFFF)
 (número_endereço = 1) AND (endereço >= 80000000) AND (endereço <= 8FFFFFFF)
 (número_endereço = 2) AND (endereço >= A0000000) AND (endereço <= CFFFFFFF)
 (número_endereço = 3) AND (endereço >= D0000000) AND (endereço <= FFFFFFFF)

Desta forma, cada evento de cobertura seria contabilizado somente no caso que, para cada evento, se cumprirem as três condições especificadas. Claramente, este é um dos motivos pelos quais, estímulos gerados aleatoriamente, devem ser coerentes em relação às dependências existentes entre eles.

2.2.3 Cobertura Funcional por Transições

Por último, encontram-se os modelos de cobertura funcional por transições, cuja vantagem é a sua grande correspondência funcional com o projeto. No caso da cobertura por transições, existe um item de cobertura para cada possível transição com a definição do conjunto de casos de teste que são a ela associados. Já existem ferramentas para a verificação por simulação, focalizadas em modelos de cobertura por transições, como, por exemplo, GOTCHA, (**G**enerator of **T**est **C**ases for **H**ardware **A**rchitectures), apresentado em (BENJAMIN et al., 1999), a qual utiliza testes de cobertura por transições como componente de um motor de simulação para verificação funcional.

2.3 Ferramentas de Verificação

A seção 2.1 se referiu aos ambientes de simulação, ou *testbenches*, como esquemas abstratos de funcionamento das ferramentas que são empregadas para executar a verificação de modelos de circuito em HDL. Na prática, existe um arcabouço (WILE; GOSS; ROESNER, 2005), esquematizado na Figura 5, que serve de base para o projeto, construção e inclusão das ferramentas de software que auxiliam no desenvolvimento dos *testbenches*.

Figura 5. Motor de trabalho da maior parte das ferramentas de simulação.

Como exposto na Figura 5, geralmente as ferramentas de verificação funcional são projetadas sobre um motor de simulação, que se encarrega de rodar o modelo em HDL do DUV em conjunto com o modelo em linguagem de verificação de hardware (*hardware verification language*, HVL) do *testbench* escrito pelo projetista de verificação. Além do motor de simulação, existem as interfaces gráficas do usuário para o controle humano da verificação, nas tarefas de aplicação e checagem de estímulos, e de monitoramento do progresso da cobertura. Estas ferramentas também devem fornecer a possibilidade de comunicação com outras ferramentas de software de geração de *testbenches*, caso o modelo em HDL previamente descrito não seja suficiente.

Existem diferentes linguagens e ferramentas que se enquadram no arcabouço da Figura 5 para realizar a verificação funcional de SoCs. Dentre as linguagens mais utilizadas para verificação funcional, encontram-se:

- **OpenVera.** É uma HVL que utiliza uma sintaxe intuitiva e combina características de HDLs, e linguagens como C++ e Java. Possui classes desenvolvidas especificamente para verificação funcional e facilita o desenvolvimento de *testbenches* e uso de asserções (SYNOPSIS, 2008).

- **e.** *e* é usado para realizar a verificação funcional de sistemas digitais usualmente descritos no nível RTL. É a primeira linguagem orientada a aspectos. As características de maior destaque desta linguagem incluem AOP (*Aspect Oriented Programming*, cuja principal característica é a separação de cada funcionalidade de código em entidades) e OOP (*Object Oriented Programming*), reuso, geração aleatória e dirigida de estímulos, com infra-estrutura e suporte integral de métricas de cobertura funcional e asserções (BERGERON, 2003).
- **SystemC.** Comumente a literatura refere-se à SystemC como uma HDL do tipo do VHDL e Verilog, mas, de fato, é uma linguagem de descrição de sistemas, pois se destaca pela sua capacidade de modelar sistemas no nível de transações e comportamental. SystemC consiste em um conjunto de rotinas e macros distribuídos em bibliotecas implementadas em C++, que possibilitam simular processos concorrentes descritos em sintaxe C++ padrão (SYSTEMC, 2008). O conjunto de bibliotecas denominadas *SystemC Verification Library*, são utilizadas para projetar *testbenches* com a mesma sintaxe C++.
- **SystemVerilog.** É a primeira linguagem de descrição e verificação de hardware (HDVL) unificada. É uma extensão da linguagem Verilog para melhorar a produtividade no projeto de SoCs baseados em IPs, de grande quantidade de portas e barramentos. A modelagem e verificação são realizadas no nível de transações, com a possibilidade de se recorrer a funções escritas em C/C++/SystemC (SYSTEMVERILOG, 2008).

Outras linguagens, como VHDL e Verilog, têm perdido aceitação para fins de verificação funcional devido às suas limitações na estruturação de *testbenches* complexos que atendam aplicações com muitos processos concorrentes.

Estas linguagens permitem verificar projetos de circuitos digitais de alta complexidade, porém, alguns dos maiores obstáculos são o grande esforço e o tempo requeridos para o seu aprendizado por parte dos engenheiros de verificação. No entanto, as ferramentas utilizadas para rodar os arquivos escritos nestas linguagens possuem uma boa funcionalidade, com capacidade de extração e análise de dados provenientes da simulação dos modelos em HDL, compensando a complexidade das HVLs. São muitas as ferramentas que suportam verificação funcional de SoCs em diferentes níveis de abstração, dentre elas destacam-se a Riviera da Aldec Inc, @Designer-PRO da Axiom, Incisive da Cadence, Questa e HDL Designer da Mentor Graphics, e VERA e CoCentric da Synopsys.

Além das ferramentas comerciais para verificação funcional mencionadas acima, existem também outras ferramentas criadas para assistir, de diversas maneiras, as tarefas de escrita de arquivos para geração de estímulos, simulação do DUV e checagem de respostas. Os autores de (JERINIĆ, 2005), (LANGER et al., 2006), (JERINIĆ; MÜLLER, 2004), e (JERINIĆ, 2007), criaram uma ferramenta como parte da metodologia detalhada na Seção 3.1. Esta ferramenta, denominada *ParaGraph*, é composta por um conjunto de bibliotecas escritas na linguagem C++ que servem para manipular grafos de domínios de parâmetros (*parameter domain graphs*, PDGs) correspondentes a qualquer tipo de projeto digital adequado a tal metodologia, facilitando a geração de casos de teste pré-determinados para, posteriormente, testar o DUV sob qualquer ambiente de simulação. Alguns aspectos mais específicos desta ferramenta serão tratados na seção seguinte.

Também, em (ROMERO, 2005), houve a implementação de uma ferramenta para auxiliar o projetista de verificação na criação de documentos dentro do formato específico para construção de *testbenches* em linguagem SystemC. Esta ferramenta, chamada de *tb_creator*, cria para o engenheiro de verificação uma versão do *testbench* que não inclui detalhes da aplicação a ser verificada, uma vez que esta terá que ser desenvolvida especificamente. Esta versão dos arquivos do *testbench*, denominada de gabarito, consiste na implementação dos módulos dos modelos do *testbench*, já conectados entre si e, internamente, com espaços livres para a inclusão dos elementos necessários para realizar a verificação.

2.4 Plano de Verificação

A verificação funcional é composta por um grande número de tarefas que devem ser executadas de forma correta e no momento apropriado. Por isto, se faz necessário elaborar um plano de verificação, ou seja, um documento que contenha todos os passos, estratégias, e recursos que permitam o sucesso da verificação. Formalmente, o plano de verificação define o que deve ser verificado e como será verificado, e deve ser atualizado permanentemente durante a etapa de verificação funcional. Ele descreve o escopo do problema de verificação para o dispositivo e serve como especificação funcional para o ambiente de verificação (PIZIALI, 2004). Os itens mais relevantes que devem ser incluídos no plano de verificação são enumerados, a seguir:

- **Especificação Funcional.** É o documento a partir do qual será estruturada a verificação inteira; é uma descrição do comportamento dos módulos e submódulos de hardware, e das funções que o dispositivo deve cumprir.
- **Ferramentas.** Todo o conjunto de pacotes de software específicos da implementação da verificação funcional deve ser especificado na íntegra no plano de verificação.
- **Metodologia Específica.** Refere-se ao modelo de verificação que servirá para depurar o código HDL elaborado pela equipe de projeto. A metodologia deve incluir aspectos como a arquitetura de verificação, modelos de *testbench*, tipos de dados, e possivelmente sugestões do modelo de cobertura.
- **Cobertura.** A cobertura é um elemento crítico do plano de verificação, já que é o ponto de referência para conhecer de forma objetiva a qualidade do processo de verificação. No caso da cobertura funcional, deve-se especificar o conjunto de estados do DUV que se deseja exercitar para obter um conjunto de respostas que serão comparadas com um modelo de referência, a partir das quais será construído um relatório sobre o progresso da cobertura. O relatório permitirá, entre outras coisas, conhecer eventualmente o nível de cobertura alcançado e o desempenho com que foi efetuada a verificação.
- **Recursos.** A equipe de verificação deve conhecer desde o começo todos os equipamentos, materiais, e demais recursos necessários para levar a cabo a verificação.
- **Cronograma.** É indispensável organizar as atividades do processo de verificação dentro de um limite razoável de tempo, já que, como foi previamente mencionado, a verificação é a etapa mais dispendiosa em termos de tempo e recursos de todo o projeto de sistemas digitais integrados.

Outros itens indispensáveis, embora aparentem ter menor relevância, consistem do estabelecimento do nível de abstração no qual será realizada a verificação, riscos e dependências que tenham de ser levadas em conta, e alternativas para todos os aspectos da verificação que possam vir a falhar ou que sejam passíveis de serem otimizados ou aperfeiçoados.

3 OTIMIZAÇÃO DO ESPAÇO DE ESTÍMULOS DE SIMULAÇÃO

O espaço de estímulos, a partir do qual são gerados os vetores de entrada ao DUV (Gerador de Estímulos, Figura 1), é passível de ser analisado e modificado, produzindo assim certo impacto no desempenho da simulação. Evidência de tal fato é o recente surgimento de trabalhos relacionados com o estudo, e otimização, do espaço de estímulos de entrada, visando melhorar diversos aspectos relativos à verificação por simulação (redundância dos estímulos, tempo de simulação, grau de cobertura atingido, nível de clareza e objetividade da informação obtida na simulação).

Neste capítulo apresentamos a metodologia desenvolvida por Vasco Jerinić em (JERINIĆ, 2005), que é uma proposta para reduzir o conjunto total de possíveis valores dos parâmetros de simulação, baseada nas dependências existentes entre eles, estabelecidas a partir da especificação funcional do projeto sob verificação; desta forma, a redução do espaço de simulação resulta numa posterior melhora da eficiência da simulação. Todo o arcabouço teórico é apresentado em detalhes, uma vez que é a base deste trabalho de mestrado. Também são apresentadas neste capítulo outras metodologias que tratam da otimização do espaço de estímulos de simulação e que tenham alguma similaridade com o trabalho em (JERINIĆ, 2005).

3.1 Domínios de Parâmetros de Verificação

A metodologia apresentada em (JERINIĆ, 2005), fundamenta-se nas dependências (inerentes à funcionalidade do dispositivo) que existem entre os parâmetros dos modelos HDL. Tais

parâmetros, que existem como mecanismo de configuração e adaptação do circuito ou sistema a diferentes aplicações, servem também como parâmetros de simulação durante a verificação dos modelos de hardware. É importante não confundir o termo ‘parâmetro’, com outros termos como ‘variável’, ou ‘sinal de entrada, de saída e interno’, pois estes são essencialmente elementos reais associados aos modelos de descrição, enquanto o primeiro termo se refere a elementos abstratos de especificação, cujos valores indicam ao sistema para realizar tarefas específicas, ou modificar seu comportamento, e são mapeados para variáveis ou sinais. Segundo (GIVARGIS; VAHID, 2000), tais parâmetros podem ser classificados em estáticos (ou estruturais, que afetam a estrutura do projeto e, portanto, devem ser declarados no nível RT) ou dinâmicos (ou funcionais, que afetam o comportamento do circuito e são armazenados em registradores de configuração).

Fundamentalmente, os parâmetros conformam um espaço, ou área, cujo tamanho está dado pelo conjunto de todos os valores que os parâmetros podem adquirir. Então, um espaço P constituído pelos parâmetros de verificação $p_1, p_2, \dots, p_n$, onde cada parâmetro pode adquirir R valores, está descrito pela Equação 1, e o seu tamanho está definido pela Equação 2 (JERINIĆ, 2007).

$$P = p_1 \times p_2 \times \dots \times p_n \quad (1)$$

$$S_{PARAM} = |P| = \prod_{i=1}^n |R_i| \quad (2)$$

De um ponto de vista analítico, o espaço P , na Equação 1, pode ser dividido em subespaços, a partir dos diferentes valores, ou conjuntos de valores, definidos para cada parâmetro na especificação funcional do sistema; tal divisão do espaço pode ser utilizada para a aplicação de estímulos aleatórios confinados (*constrained random stimuli*) e abre o caminho, como se mostrará ainda neste capítulo, para definir o conceito de *Domínios de Parâmetros*, e aplicá-lo à simulação e otimização da verificação funcional. A Figura 6 mostra um exemplo de um espaço dividido em vários destes subespaços, compostos pelo conjunto de todos os possíveis vetores de estimulação do sistema.

Quando vista a partir desta perspectiva de subespaços de valores, a estimulação aleatória pode, em muitas ocasiões, ser fonte de valores pertencentes a subespaços com redundância ou sem interesse, como definidos a seguir.

- Definição 1. Para a simulação de um sistema, subespaços determinados por parâmetros de simulação são redundantes quando possuem uma área comum de valores de estimulação, representando assim o mesmo evento de cobertura funcional.
- Definição 2. Para a simulação de um sistema, subespaços determinados por parâmetros de simulação são inválidos quando formados por valores que produzem, segundo a especificação funcional, condições ilegais ou indeterminadas no funcionamento do sistema.

Na Figura 6, por exemplo, dois conjuntos diferentes de simulação formados pelos três parâmetros de simulação *param1*, *param2*, e *param3*, possuem um espaço comum que se traduz em redundância, ou seja, valores que representam o mesmo evento de cobertura funcional, portanto, desnecessários de serem aplicados uma segunda vez, se for para aumentar a eficiência dos estímulos gerados. Pode-se observar também na figura que existem dois conjuntos de estímulos que representam condições não válidas segundo a especificação do sistema, sendo necessária sua exclusão do espaço de simulação. Na figura, o subespaço A engloba parte do subespaço inválido 2, portanto engloba condições que não são de interesse para a simulação.

Figura 6. Espaços de simulação sobrepostos.

Resumindo, para prevenir que valores redundantes, ou valores inválidos, sejam gerados, e assim reduzir o tempo necessário para realizar a simulação, duas condições devem ser mantidas:

- somente devem ser usados subespaços que não se sobreponham;
- combinações inválidas de parâmetros devem ser excluídas do espaço de simulação.

3.1.1 Definição de Domínios de Parâmetros

Relacionado ao problema de espaços sobrepostos ou configurações inválidas dos parâmetros de simulação, define-se o conceito de *Domínios de Parâmetros* (BERGERON, 2003):

“Um Domínio de Parâmetros é um número de combinações válidas de alocação de parâmetros que descreve de modo único um conjunto de valores para cada parâmetro, sendo por sua vez, combinações únicas no espaço total de parâmetros”.

Se os valores para os parâmetros de simulação forem escolhidos a partir de domínios de parâmetros, então, a sua validade estará garantida, e subespaços sobrepostos excluídos. A Figura 7 apresenta uma representação gráfica do que poderia ser um exemplo de espaço de simulação reduzido e organizado, de acordo com a definição de domínios, e com a exclusão de valores inválidos e redundantes.

Figura 7. Subespaços dos parâmetros de simulação que formam Domínios de Parâmetros.

3.1.2 Modelo de Dependência

Para reduzir a quantidade de valores que os parâmetros podem adquirir é necessário deduzir todas as relações de dependência existentes entre estes parâmetros. Tais dependências devem ser extraídas do documento de especificação funcional do projeto sob verificação (GIVARGIS; VAHID, 2000). Na modelagem para representação de dependências entre parâmetros em (JERINIĆ, 2005), utiliza-se os *Grafos de Domínios de Parâmetros* (PDGs) para expor de maneira gráfica os domínios de parâmetros. Um PDG é, em termos gerais, uma árvore, extensão dos diagramas de decisão múltipla (MDDs) (LANGER et al., 2006), onde os nós e ramos são utilizados para armazenar e mostrar domínios de parâmetros e seus valores permitidos.

As relações de dependência podem ser de dois tipos, segundo a forma como uns parâmetros influem sobre os outros. Primeiro, a relação do tipo *conflito* estabelece que:

“Um parâmetro X não tem efeito sobre o comportamento do DUV caso outro parâmetro Y assuma um determinado valor ou conjunto de valores, ou seja, X tem um CONFLITO com o valor de Y”.

Uma segunda relação é o do tipo *requisito* e estabelece que:

“Um parâmetro X influirá no comportamento do DUV apenas se outro parâmetro Y assumir um determinado valor ou conjunto de valores, ou seja, X tem um REQUISITO do valor de Y”.

Como exemplo ilustrativo, consideremos que fosse necessário reduzir o espaço de simulação de um IP que contém os parâmetros (com todos seus possíveis valores entre colchetes):

- *paridade* [*par*, *impar*, *off*]
- *posição* [1.. 10]
- *comprimento* [1.. 10]

O parâmetro *paridade* define o tipo de paridade (par ou impar) a ser usado na simulação, ou, então, se a geração de paridade está desligada (*off*). O parâmetro *posição* serve para escolher a posição do bit de paridade dentro do pacote enviado, enquanto o parâmetro *comprimento*

regula o número de bits utilizados para formar o pacote de informação. Então, a cardinalidade do espaço de parâmetros, segundo a Equação 2, é:

$$|P| = \prod_{i=1}^n |R_i| = R_{paridade} \times R_{posição} \times R_{comprimento} = 3 \times 10 \times 10 = 300$$

Não obstante, existe uma relação de *CONFLITO* entre os parâmetros *paridade* e *posição* que pode ser aproveitada para reduzir o espaço de parâmetros de simulação; no caso em que o parâmetro *paridade* assuma o valor *off*, o parâmetro *posição* não tem efeito nenhum sobre o DUV. Deste modo, criam-se neste exemplo três domínios de parâmetros:

- *paridade[par], posição[1..10], comprimento[1..10]*
- *paridade[impar], posição[1..10], comprimento[1..10]*
- *paridade[off], comprimento[1..10]*

A cardinalidade do novo espaço pode ser obtida somando o número de valores permitidos de cada domínio criado:

$$|P| = D_{paridade \Rightarrow par} + D_{paridade \Rightarrow impar} + D_{paridade \Rightarrow off} = 1 \times 10 \times 10 + 1 \times 10 \times 10 + 1 \times 10 = 210$$

Obtém-se, então, uma redução de 30% no número de casos de teste que devem ser aplicados ao DUV durante a simulação. As Figura 8a e Figura 8b mostram, através de PDGs, uma representação gráfica do exemplo anterior.

Observe-se que uma relação de conflito pode ser descrita como de requisito, e vice-versa, porém com diferentes graus de complexidade; no exemplo anterior, por exemplo, a relação de requisito exigiria que a paridade “*par*” e “*ímpar*” fossem descritos explicitamente. Assim, a dualidade entre os dois tipos de dependências pode ser vista, como pseudocódigo, da forma seguinte:

CONFLITO:

parâmetro ‘posição’
com parâmetro ‘paridade == off’

REQUISITO:

parâmetro ‘posição’
com parâmetro ‘paridade == par’
‘paridade == impar’

Figura 8a. Espaço de simulação sem levar em conta dependências entre parâmetros.

Figura 8b. MDD representando o novo espaço de simulação reduzido através das dependências entre os parâmetros.

Apesar de que nesta dissertação, apenas os casos de conflito e requisito foram necessários nos casos de estudo, existem outras formas de dependência entre parâmetros. Em (JERINIĆ; MÜLLER, 2004), algumas relações de tipo lógico-aritmético foram adicionadas para se declarar dependências entre parâmetros que não podem ser expressas apenas por conflitos ou requisitos. Por exemplo, a relação a seguir estabelece que `parâmetro_1` sempre deverá ser menor do que a soma de `parâmetro_2` e `parâmetro_3`:

$$\text{parâmetro_1} < \text{parâmetro_2} + \text{parâmetro_3}$$

Adicionalmente, os valores de um parâmetro podem ser truncados por operadores lógicos do tipo AND e OR. O exemplo a seguir mostra que quando o `parâmetro_1` assume o valor '0', o `parâmetro_2` deve ser menor ou igual ao `parâmetro_3`, mas, quando o `parâmetro_1` é igual a '1', a restrição nos valores do `parâmetro_2` desaparece:

```
{ (parâmetro_1 == 0) && [parâmetro_2 <= (parâmetro_3)] || (parâmetro_1 == 1)
```

Os valores dos parâmetros de simulação podem ser limitados ainda mais através de equações, ou seja, relações de igualdade que designam valores específicos aos parâmetros. Exemplos deste tipo de dependência seriam:

$$\begin{aligned} \text{parâmetro_1} &= 3 * \text{parâmetro_2} \\ \text{ou} \\ \text{parâmetro_1} &= (\text{parâmetro_2} \% \text{parâmetro_3}) ^ 2 \end{aligned}$$

Do ponto de vista da utilização dos PDGs para o gerenciamento da estimulação, dois aspectos podem ser destacados:

- no trabalho em (JERINIĆ, 2005), todo o conjunto de valores dos parâmetros é usado para a estimulação direta; em circuitos mais complexos como os utilizados como caso de estudo nesta dissertação, isto não é possível, portanto um esquema de estimulação aleatória é adotado.
- além dos parâmetros de interesse que ocorrem no PDG, existem outros sinais de entrada que devem ser estimulados durante a verificação; para estes, a geração seria puramente aleatória.

3.2 Outros Trabalhos Relacionados

De forma similar ao arcabouço teórico exposto na seção anterior, existe uma preocupação generalizada por produzir metodologias otimizadas de geração automática de estímulos objetivando cobrir a funcionalidade completa do projeto, com maior velocidade, e conseguir uma eficiência maior da verificação. Tem surgido recentemente trabalhos com metodologias que afetam o espaço original de valores dos parâmetros de simulação. A seguir, são apresentados resumidamente alguns destes trabalhos, para assim, descrever o impacto que determinadas estratégias de geração de estímulos podem ter sobre o desempenho da verificação.

3.2.1 Verificação Funcional de Descrições SystemC usando Solução de Restrições (FERRANDI; RENDINE; SCIUTO, 2002)

Este trabalho propõe a utilização de algoritmos que geram vetores de teste automaticamente (ATPG, *Automatic Test Pattern Generation*) para sistemas descritos como máquinas de

estados finitos (FSM, *Finite State Machine*). Trata-se de uma proposta de implementação de algoritmos de ATPG baseados em controlabilidade e observabilidade, cujo principal objetivo é solucionar o problema de como gerar um conjunto de vetores de teste de entrada, suficientemente preciso, para cobrir a maior quantidade possível de funções do projeto, sob restrições de recursos, geralmente representados pelo tempo disponível para realizar a verificação.

O método de geração proposto em (FERRANDI; RENDINE; SCIUTO, 2002) é realizado através de cinco tarefas principais:

- aquisição de dados: organização de toda informação disponível sobre o sistema em listas de portas de entrada, expressões, instruções condicionais, e transições. Tais listas são obtidas através de uma análise do código fonte em SystemC.
- análise de transições: consiste em achar os estados inicial e final para cada transição presente na lista de transições.
- enumeração de seqüências: enumeração de caminhos (seqüências) potenciais de execução, no programa escrito em SystemC, para os quais são requeridos os vetores de teste.
- análise de seqüências e produção de restrições: nesta etapa do algoritmo é produzido um conjunto de restrições, para cada seqüência, correspondentes a todas as instruções condicionais que devem resultar verdadeiras ao longo da execução da seqüência. Devido ao fato de tais restrições não dependerem somente das portas de entrada, mas também de variáveis e sinais, é realizado um rastreamento reverso no código para obter uma dependência direta de tais variáveis e sinais em função das portas de entrada. O resultado desta etapa é um conjunto de equações que correspondem a cada restrição.
- solução de restrições e extração de testes: finalmente, deve-se gerar uma solução para satisfazer as restrições associadas a cada seqüência. Depois, os vetores de teste são extraídos da solução anterior, e arranjados em um formato apropriados para o simulador SystemC.

Esta técnica baseada em ATPG resulta em uma geração de estímulos em forma direta, como visto na Seção 2.1.1. Segundo os resultados, este trabalho aparenta um desempenho melhor do

que outras técnicas baseadas em ATPG, porém tem duas desvantagens importantes: primeiro, trata-se de uma metodologia construída estritamente sobre padrões de cobertura estrutural, ou de código, que possui pouca relação funcional com o projeto sob verificação, e, segundo, ela não consegue garantir cem por cento de cobertura mesmo para sistemas de complexidade média.

3.2.2 Seleção de Testes Funcionais baseada em Análise de Vetores de Suporte Não-Supervisados (GUZEY et al., 2008)

Nos últimos anos, as máquinas de vetores de suporte (*Support Vector Machines*, SVMs) têm se tornado um dos métodos de classificação de dados mais populares, tendo sido utilizadas em uma grande variedade de aplicações tais como classificação de texto, reconhecimento de expressão facial, e análise de genes (FRADKIN; MUCHNIK, 2006). Os SVMs podem ser entendidos como um método para a construção de um tipo especial de regra, chamada de classificação linear, de forma que novos classificadores, ao serem produzidos, garantem um bom desempenho preditivo (alta qualidade de classificação nos dados ainda não analisados). O fundamento teórico deste método é fornecido pela teoria do aprendizado estatístico (VAPNIK, 1998).

A capacidade preditiva das máquinas de vetores de suporte pode ser utilizada em verificação funcional para, a partir do espaço completo de estímulos de entrada, estimar um subespaço constituído por um conjunto específico de testes que maximizam a cobertura funcional. Isto permite reduzir o número de ciclos de simulação requeridos para alcançar uma dada meta de cobertura. Para fazer a estimativa, a metodologia em (GUZEY et al., 2008) se baseia na similaridade existente entre diferentes estímulos. A similaridade é analisada através de uma função de similaridade k , de forma que $k(t_i, t_j)$ mede a similaridade entre os vetores de teste t_i e t_j . Assim, supondo que, por exemplo, existem três vetores de teste t_1 , t_2 , e t_3 , e fosse possível achar uma função k tal que $k(t_1, t_2) > k(t_1, t_3)$, então, o espaço de cobertura funcional alcançado ao utilizar os testes t_1 e t_2 na simulação será, provavelmente, também maior do que o espaço de cobertura correspondente aos testes t_1 e t_3 .

A Figura 9 representa graficamente a idéia de se selecionar e agrupar os casos de teste que compõem o espaço de simulação em base à sua similaridade. A figura exemplifica um espaço

de 21 testes, para o qual é possível realizar agrupamentos desenhando-se seis círculos de radio R , sendo que todos os testes dentro de um círculo são mais similares entre si do que com testes fora do círculo. Se os testes dentro de cada círculo forem suficientemente similares, então, seria possível reduzir o espaço de estímulos, de 21, para 6 testes, escolhendo-se um de cada dos 6 círculos.

Figura 9. Seleção de testes importantes baseando-se em medições relativas de similaridade.

O problema de agrupar e escolher os indivíduos (testes, representados por pontos na figura), é categorizado como um problema de aprendizado não supervisionado (VAPNIK, 1998). A escolha dos testes mais efetivos do espaço de total de estímulos é realizada da forma seguinte:

1. A partir da uma função de similaridade k (comumente são usadas as funções produto cruz, ou gaussiana (GUZEY et al., 2008)), e um conjunto arbitrário de testes T_1 , é gerado um modelo M_T , que contém a informação de similaridade dos testes em T_1 .
2. O modelo M_T é utilizado como filtro para excluir de um segundo conjunto de testes T_2 , todos os testes similares àqueles contidos em T_1 , produzindo assim um subconjunto de testes T_{s2} .
3. É realizada uma simulação do sistema utilizando os conjuntos $(T_1 + T_2)$ e $(T_1 + T_{s2})$, um após o outro. Se a cobertura atingida por cada conjunto de testes for similar, se realiza mais uma iteração do algoritmo começando com T_{s2} no lugar de T_2 . Finalmente, é obtido um conjunto de testes reduzido que economiza ciclos de simulação.

No final do processo iterativo anterior, é obtido como resultado um conjunto de testes de redundância altamente reduzida; sendo um processo algorítmico não determinístico, não é possível garantir redundância nula. Finalmente, já que a função de similaridade inclui a informação de restrições dos valores de simulação (para excluir valores inválidos e reduzir a

redundância antes do início algoritmo de aprendizado), o sistema pode ser simulado gerando estímulos aleatoriamente a partir daquele conjunto de testes reduzido.

Com a metodologia mencionada, os autores de (GUZEY et al., 2008) conseguem obter melhorias significativas no tempo de simulação, aplicado a verificação do OpenSPARC T1, microprocessador de 64 bits de código aberto desenvolvido pela SUN Microsystems (OPENSPARC, 2009). Considerados os resultados apresentados pelos autores deste último trabalho, seria possível realizar uma comparação (em termos de reprodutibilidade, desempenho, objetividade, e nível de relação com a cobertura funcional) com a metodologia de verificação funcional através de domínios de parâmetros (Seção 3.1). Porém, tal comparação requereria um estudo detalhado da aplicação da metodologia de verificação com vetores de suporte, o que não foi possível, dada as restrições de tempo. Portanto, tal atividade fica proposta como um dos trabalhos a serem desenvolvidos futuramente.

3.2.3 Otimização da Verificação Dirigida por Cobertura e Análise de Casos Extremos usando Diagramas de Decisão (LANGER et al., 2006)

Este trabalho foi desenvolvido pelos mesmos autores de (JERINIĆ, 2005) e (JERINIĆ; MÜLLER, 2004) baseando-se no desenvolvimento prévio da metodologia de verificação a partir de PDGs, que parece ter um melhor desempenho do que outras abordagens mais convencionais relacionadas à geração aleatória de estímulos. O objetivo principal, neste trabalho, é fazer dar ênfase nos casos extremos, definindo-os a partir das dependências entre os parâmetros de verificação, como na Seção 3.1.2.

Os resultados obtidos em (LANGER et al., 2006) são ilustrados de forma geral pela Figura 10. A linha reta (1) na Figura 10 mostra o desempenho da verificação obtido em (LANGER et al., 2006) utilizando PDGs para aplicar esta ênfase especial de casos extremos na cobertura funcional, e para diminuir o espaço de verificação. A linha que parece ter um comportamento exponencial assintótico (2) utiliza também geração aleatória de casos de teste que se encontram dentro do espaço definido por um PDG, mas sem nenhum tipo de ênfase ou classificação especial na cobertura funcional do *testbench*. Finalmente, a linha inferior (3) representa o progresso da cobertura para caso de estimulação completamente aleatória. Aparentemente, é a técnica que possui o desempenho mais pobre.

Figura 10. Comparação entre enfoques diferentes de cobertura.

Analisando as curvas da evolução da cobertura na Figura 10, a estimulação direta orientada por cobertura de casos extremos mostra ser a melhor alternativa para realizar a verificação funcional do sistema que foi utilizado como exemplo neste trabalho. Porém, para o caso dos sistemas digitais comerciais atuais (que possuem uma complexidade bastante maior do que o exemplo mencionado), é necessário reconsiderar o uso de alternativas de estimulação aleatória para cobrir os grandes espaços de verificação característicos de tais sistemas.

No âmbito da presente dissertação, adotou-se um esquema de simulação aleatória a partir de domínios de parâmetros, para demonstrar sua real vantagem sobre a verificação de modelos de sistemas de grandes dimensões, como os que serão apresentados no Capítulo 5.

4 METODOLOGIA DE VERIFICAÇÃO COM PDGS

O quadro conceitual específico do presente trabalho foi estabelecido na Seção 3.1, onde foram definidos os *Grafos de Domínios de Parâmetros* (PDGs) e sua utilidade para representar mediante eles um “espaço efetivo” de estímulos. Este tipo especial de estrutura de representação de dados (neste caso, parâmetros de simulação e seus valores) é o núcleo da metodologia aqui apresentada. No entanto, a complexidade para se estruturar um PDG (baseada na complexidade dos modelos HDL comerciais atuais) requer o uso de ferramentas de software e exige bons recursos computacionais para se conseguir o aproveitamento e manipulação das dependências a serem extraídas da especificação funcional do DUV.

A ferramenta *ParaGraph* (JERINIĆ, 2005) é integrada por um conjunto de bibliotecas escritas na linguagem C++ e projetadas especificamente para a construção e manipulação de diferentes tipos de diagramas de decisão, inclusive e especialmente, PDGs. Para isto, as bibliotecas fornecem todo um conjunto de funções que auxiliam o processo de interpretar as dependências entre os parâmetros, construir o diagrama correspondente, e eventualmente, permitir a automação na geração de estímulos a serem aplicados ao DUV. Tais estímulos correspondem aos vetores de valores da cada caso de teste pertencente ao PDG obtido, que podem estar num arquivo de código compatível com a linguagem de verificação escolhida pelo projetista (em nosso caso, SystemC).

Neste capítulo apresentamos alguns detalhes do *ParaGraph*, além de uma pequena descrição da biblioteca de verificação em SystemC (SYSTEMC, 2008) (*SystemC Verification Library*, SCV), cujos elementos são utilizados na geração de estímulos dos *testbenches*. A seguir, são

apresentadas uma seqüência de passos da qual consiste a metodologia para a verificação com PDGs e uma ferramenta para a geração de estímulos em SystemC a partir de PDGs.

4.1 O Ambiente de Visualização Gráfica e Checagem de Parâmetros:

ParaGraph

Como mencionado na Seção 2.3, os autores de (JERINIĆ, 2005) desenvolveram uma ferramenta visando melhorar e facilitar a simulação dos modelos HDL de sistemas digitais. Esta ferramenta, chamada *ParaGraph* (abreviatura das palavras em inglês *Parameter Checking and Graphical Visualization*), está integrada por um conjunto de bibliotecas de funções básicas escritas na linguagem C++, e vários componentes de execução de código interpretado (não compilado), na linguagem Python, para a criação, visualização, e manipulação de grafos de domínios de parâmetros. Os autores disponibilizaram a ferramenta para possibilitar a realização deste trabalho de mestrado. Nas sub-seções seguintes, são descritas várias características e aspectos relacionados à ferramenta *ParaGraph*, relevantes para a presente dissertação.

4.1.1 Implementação das Funções das Bibliotecas *ParaGraph*

Os autores desta ferramenta utilizaram uma estrutura hierárquica de objetos, estruturados em classes, para dar formato e organizar as múltiplas funções que compõem o ambiente. A Figura 11 ilustra de forma simplificada esta hierarquia de classes e subclasses da ferramenta. A classe de diagramas de decisão contém funções diversas para realizar todo tipo de operações sobre diagramas binários, diagramas de decisão múltipla, e diagramas híbridos. A classe de utilidades inclui ferramentas para ordenamento de dados, operações matemáticas complementares, e análise de validade de objetos representados por diagramas de decisão. A classe para grafos de domínios é composta por tipos de dados, operadores, e funções específicas para manipular os PDGs. Os elementos mais representativos desta última classe, incluindo subclasses e funções, são:

– subclasses:

struct Dependency: dependência entre dois parâmetros, ou valores de parâmetros.

Figura 11. Estrutura hierárquica de classes do *ParaGraph*.

struct Domain: uma folha dentro do *DomainGraph*.

struct DomainGraph: classe de mais alto nível *DomainGraph*.

struct Edge: um ramo dentro do *DomainGraph*.

struct Node: um nó dentro do grafo.

– funções:

read_definitions: ler dependências desde arquivo de texto em formato *.def* e criar o *DomainGraph*.

write_definitions: escrever o *DomainGraph* de novo num arquivo de texto *.def*.

create_graphs: construir um grafo (chamada desde *read_definitions*).

create_domains: criar todos os domínios do grafo (chamada desde *read_definitions*).

read_pdg: ler um *DomainGraph* no formato gerado pelas funções anteriores (utilizada internamente).

Todos os tipos de dados, classes, funções, e operadores definidos nas bibliotecas do *ParaGraph* podem ser instanciados dentro de arquivos de código C++ quaisquer, no entanto, para fins de visualização e facilidade de manipulação dos PDGs, foram incluídos na

ferramenta métodos de compilação, que permitem importar as bibliotecas dentro de scripts de código escrito na linguagem Python. Alguns detalhes são apresentados a seguir.

4.1.2 Interface Python para Controle e Visualização de PDGs

Python é uma linguagem orientada a objetos desenvolvida pela *Python Software Foundation* (PYTHON, 2008). Caracteriza-se por ser uma linguagem interpretada, não compilada como C, C++, C#, etc. Por este motivo, os programas escritos em Python costumam ser chamados de scripts; estes scripts, uma vez escritos, podem ser executados através do interpretador Python, da mesma forma que um arquivo binário qualquer.

Uma das maiores vantagens desta linguagem é a capacidade de extensão a partir e para outras linguagens de programação, sendo assim possível, por exemplo, embutir código escrito em C++ dentro de scripts Python, e vice-versa, através de diferentes interfaces de programação de aplicativos (*Application Programming Interfaces*, ou APIs). Aproveitando esta característica, os autores do *ParaGraph* projetaram uma interface básica escrita em Python (isto é, um script) que utiliza as bibliotecas originariamente escritas em C++. O objetivo desta implementação foi permitir o uso dos recursos de criação de interfaces gráficas (para visualização de PDGs), e alto nível de abstração (adequado às classes *ParaGraph*), próprias de tal linguagem, integrando-a com as bibliotecas do ambiente *ParaGraph*. Tal interface permite, dentro de um script Python, a leitura das dependências entre parâmetros de um sistema, criação do PDG correspondente e sua visualização de forma gráfica², e realização de qualquer tipo de operação com o objeto que o representa.

Dentro do quadro de trabalho da presente dissertação, uma nova interface foi projetada, baseada naquela interface inicial desenvolvida pelos autores da suíte *ParaGraph*. Esta nova interface, consistente também de um script Python, possibilita a criação de geradores de estímulos aleatórios baseados em domínios de parâmetros, na linguagem SystemC. Os detalhes desta nova ferramenta, chamada *GET_PRG*, são apresentados na Seção 4.4.

4.1.3 Especificação de Parâmetros de Simulação

² A representação gráfica dos PDGs gerados pelos scripts incluídos na ferramenta *ParaGraph* têm um propósito meramente descritivo e qualitativo, em relação às dependências entre os parâmetros de simulação de um sistema, portanto, não são consideradas fundamentais para o presente trabalho.

O *ParaGraph* permite que se estabeleçam todos os parâmetros de simulação, e, também, o conjunto completo de valores definido para cada um deles na especificação funcional do DUV. Os parâmetros devem ser declarados em uma sintaxe simples para serem interpretados pela ferramenta *ParaGraph*, podendo pertencer às seguintes categorias (BENJAMIN et al., 1999):

- **STATIC**. Parâmetros cujos valores devem ser estabelecidos previamente à fabricação do SoC. Assim, os valores destes parâmetros são fixos em relação ao SOC fabricado. Normalmente estes valores aparecem no código HDL (estrutural ou comportamental) usado para, eventualmente, criarem instâncias do SOC. Em geral, estes parâmetros são declarados em sentenças *generic* no código HDL, como por exemplo, o tamanho do buffer para uma UART.
- **DYNAMIC**. Parâmetros que afetam o comportamento do SOC, e que são reconfigurados após sua fabricação. Por exemplo, parâmetros que indicam transições nos sinais ou registradores, nível de ocupação de buffers, dados de entrada ou saída, etc.
- **SIGNAL**. Parâmetros que assumem os seus valores a partir de sinais de transmissão de dados. Geralmente correspondem a portos de entrada ou saída de dados.

Para especificar os limites dos valores que os parâmetros podem adquirir, empregam-se também as palavras chave **RANGE**, para designar um conjunto específico de valores, e **MAP**, para associar valores específicos a nomes que descrevem sua influência ou comportamento no DUV (JERINIĆ, 2005).

4.2 SystemC Verification Library, SCV

SystemC é uma linguagem de descrição de sistemas já madura, constituída por um conjunto de bibliotecas implementadas em C++, possibilitando a simulação de processos concorrentes sob a sintaxe C++ convencional. Além das bibliotecas destinadas para especificação de sistemas digitais, a linguagem SystemC inclui também bibliotecas cuja finalidade é realizar a verificação de tais sistemas, denominadas de SystemC Verification Library (SCV) (SYSTEMC, 2008). As características principais deste padrão de verificação da SystemC são:

- tratamento de tipos de dados arbitrários.
- verificação no nível de transações.
- geração aleatória de estímulos.
- restrições para a geração aleatória.
- ponderação da geração aleatória.

Tais características podem ser consideradas vantajosas por agilizar a projeção de código dos *testbenches*, e facilitar a análise de resultados de simulação (pelo nível de abstração em transações). Neste trabalho, as bibliotecas SCV foram utilizadas para projetar os geradores de estímulos para verificação dos casos de estudo mostrados, mais adiante, no Capítulo 5. De forma específica, os objetos das bibliotecas SCV que fizeram parte de nossa implementação são:

- `scv_bag< pair <int,int> >` distribuição: declaração de distribuição de valores de estímulos; inicialmente esta distribuição está vazia.
- `distribuição.push(pair<int,int>(n1,n2),Q)`: adição de Q valores à distribuição anterior, que se estendem desde $n1$ até $n2$. É possível adicionar vários conjuntos de valores à distribuição.
- `parâmetro.set_mode(distribuição)`: estabelece que o parâmetro pode adquirir apenas os valores aleatórios segundo os existentes em `distribuição`. Neste trabalho, é utilizado para restringir os valores dos parâmetros para cada domínio.
- `distribuição.clear()`: apagamento de todos os valores previamente adicionados à distribuição. Útil para mudar as restrições dos parâmetros dinamicamente durante a simulação.

Maiores detalhes a respeito do uso destes objetos podem ser consultados em (SYSTEMC, 2008). Também, em (DA SILVA; MELCHER; ARAUJO, 2004), são apresentados alguns aspectos específicos do uso das bibliotecas SCV para verificação funcional de sistemas sob um modelo de *testbench* como aquele apresentado na Figura 1. Na Seção 4.4 é mostrada a forma com que nossa ferramenta, o *GET_PRG*, utiliza as funções incluídas nas bibliotecas SCV para automatizar o código de geração de estímulos a partir de PDGs.

4.3 Geração do PDG e criação do Gerador de Estímulos em SystemC

A metodologia para a geração do PDG e, posteriormente, a ferramenta *GET_PRG* para a obtenção do gerador de estímulos em SystemC são ilustradas na Figura 12. A seguir, uma descrição da sequência algorítmica de todo o processo é feita.

Inicialmente, como colocado na Seção 3.1.2, é necessário extrair todas as dependências entre parâmetros a partir do documento de especificação funcional do DUV, e organizá-las sob as diretrizes do formato *ParaGraph*. Estas dependências, organizadas e escritas manualmente em um arquivo de definição de dependências (arquivo com extensão *.def*, segundo os requisitos das bibliotecas *ParaGraph*) são lidas pela ferramenta para, seguidamente, serem interpretadas e uma conversão do arquivo de texto de dependências, para uma representação de dados interna, o PDG, é feita. É possível então visualizar o PDG correspondente, acrescentar ou apagar nós e ramos de forma manual, e de forma geral, realizar qualquer tipo de operação com as funções incluídas nas bibliotecas do *ParaGraph*.

Em princípio, não há a necessidade de modificar o PDG gerado para um dado conjunto de parâmetros, ainda que seja possível realizar operações manuais sobre os objetos das estruturas de dados que representam tais PDGs. Porém, esta funcionalidade de manipulação do PDG não foi incluída na ferramenta Python dado que não é um objetivo fundamental dentro do presente quadro de trabalho.

Posteriormente também é possível criar um arquivo de texto que inclui todos os parâmetros de verificação e os valores que cada um deles deve adquirir, numa sequência específica para serem transmitidos ao DUV (através do *Driver*, Figura 1) na simulação. Este arquivo de texto, consistente de um *header* na linguagem SystemC (arquivo com extensão *.h*), foi gerado, para cada um dos casos de modelos introduzidos no Capítulo 5, com ajuda da ferramenta de software *GET_PRG* que, como mencionado, foi projetada na linguagem Python.

Figura 12. Algoritmo do script Python para criação de geradores de estímulos SystemC.

4.4 A Ferramenta *GET_PRG* (*Generation Tool for Parameter-Domain-based Random Stimuli Generation*)

Uma vez que o ambiente *ParaGraph* gera o PDG internamente, o *GET_PRG* executa uma varredura do PDG, domínio por domínio; por sua vez, cada domínio completo é percorrido, sendo cada domínio uma trajetória a partir de um nó de grau zero até o nó que se encontra no nível superior da árvore. Tal ordem de percurso permite ao script *GET_PRG* realizar a escrita do código SystemC de cada domínio individual, a começar pelos valores, ou conjuntos de valores, dos parâmetros que se encontram no nível mais baixo do PDG. O seguinte pseudocódigo é uma versão simplificada da estratégia adotada para percorrer os PDGs, numa ordem pré-determinada, e escrever o código correspondente no gerador de estímulos em SystemC:

```

for (i = 0 ; i < quantidade_domínios ;i++)
{

    // Adquirir referência do primeiro parâmetro do primeiro domínio:
    parametro = domínio[i].parametro_inferior

    escrever("  while(1) {")    // Laço de execução global e contínua do
                                // gerador de estímulos

    for (j = 0 ; j < quantidade_parametros ;j++)
    {
        // Definições dos valores de parametros para cada domínio:

        escrever( nome_parametro[j] + "_distrib.clear();" )
        escrever( nome_parametro[j] + "_distrib.push(pair<int,int> (" +
            valores_parametro[j] + "), numero_de_valores )"
        escrever( nome_parametro[j] + ".set_mode(" + nome_parametro[j] +
            "_distrib )" )

        // Avançar ao parâmetro seguinte:
        parâmetro = parametro.pai
    }

    // Laço de geração aleatória de cada domínio:
    escrever("for (int domsize = 0 ; domsize < " +
        tamanho(domínio[i]) + " ; domsize++) {"

    // Valor aleatório seguinte dentro do domínio atual:
    escrever("trans_constraint.next();" +
        "tr_ptr = new trans_struct(trans_constraint.read());" )

    // Escrita do novo valor aleatório
    escrever("estimulo.write( tr_ptr ) )"

    escrever("  }") // Finalização de laço local de domínio

    escrever("}") // Finalização de laço global de execução do gerador
}

```

A estrutura final do código SystemC escrito pela ferramenta *GET_PRG* consistirá, conseqüentemente, de vários laços de geração de estímulos aninhados em um laço de execução global e contínua. Cada um destes laços corresponde, assim, a cada um dos domínios gerados pelo ambiente *ParaGraph*.

São dois os motivos pelos quais o *GET_PRG* foi projetado para criar ciclos de várias iterações para cada domínio. O primeiro, é que, mesmo organizando os parâmetros de simulação em domínios, o espaço de valores de cada domínio resulta ser demasiado grande para que uma simulação totalmente dirigida possa ser realizada, portanto, sendo necessária uma estimulação aleatória dentro de cada domínio. A segunda razão é a diferença dos tamanhos dos domínios, ou seja, uma vez que os domínios sempre terão tamanhos diferentes entre si, é preciso estimular cada domínio de forma proporcional ao seu tamanho; esta estimulação proporcional não seria possível se, por exemplo, o gerador enviasse um só estímulo para cada domínio dentro de um ciclo global de execução.

Observando o pseudocódigo anterior, é possível perceber que, para iniciar o laço de execução local de cada domínio, emprega-se a função “`tamanho(dominio[i])`”. A necessidade do uso de tal função é devido, como exposto acima, aos diferentes tamanhos dos domínios resultantes da leitura das dependências entre os parâmetros do sistema. Tal função, por sua vez, faz uma chamada à função de cardinalidade (*cardinality*), que se encontra pré-definida nas bibliotecas do *ParaGraph*, e serve para calcular o número total de estímulos que compõem cada domínio. Deste modo, a função “`tamanho`” executa a seguinte seqüência de passos para estabelecer o número de iterações para cada domínio:

1. cálculo do número de estímulos de cada domínio usando a função *cardinality*.
2. cálculo da média e variância do tamanho numérico de todos os domínios.
3. alocação de um número de iterações para cada domínio, que dependerá, em ordem de prevalência, do tamanho inicial de cada domínio, da média, e da variância calculadas.

Embora a técnica dos PDGs se mostre vantajosa para reduzir a redundância dos estímulos de simulação, sua efetividade está fortemente vinculada à existência de dependências entre os parâmetros do sistema. Esta é uma limitação intrínseca da metodologia de verificação a partir de domínios de parâmetros, já que, quando tais dependências forem inexistentes, o espaço de

simulação será o espaço completo de estímulos de simulação. Em relação à ferramenta aqui desenvolvida, os parâmetros que forem independentes dos outros parâmetros, são declarados da forma tradicional dentro do gerador de estímulos SystemC. Assim, no pior dos casos, a ferramenta criará um gerador de estímulos completamente convencional, sendo esta uma alternativa de verificação não intrusiva.

5 DOIS CASOS DE ESTUDO: UM ADAPTADOR *BLUETOOTH* BANDA BASE E UM DECODIFICADOR *MPEG-4*

Neste capítulo apresentam-se os detalhes, tanto da especificação funcional, quanto da implementação dos modelos dos circuitos utilizados para aplicar, e analisar, a metodologia de verificação com domínios de parâmetros (Capítulo 4). O primeiro, desenvolvido pelo Grupo de Projeto de Sistemas Eletrônicos Integrados e Software Aplicado, Grupo SEIS, do Laboratório de Microeletrônica da USP, consiste de um controlador de rede sem fio no protocolo *Bluetooth*. O segundo circuito, um decodificador de vídeo no padrão *MPEG-4*, foi desenvolvido no Departamento de Sistemas da Computação da Faculdade de Engenharia da Universidade Federal de Campina Grande.

Tanto no caso do controlador *Bluetooth*, quanto no caso do decodificador de vídeo *MPEG-4*, trata-se de sistemas compostos por diferentes submódulos que executam tarefas especializadas. Após o detalhamento dos aspectos estruturais e funcionais dos diversos módulos e submódulos, os parâmetros correspondentes, utilizados dentro da metodologia apresentada no capítulo anterior, são descritos, assim como as suas dependências.

5.1 Controlador da Camada Banda Base do Protocolo *Bluetooth*

O *Bluetooth* (versão 2.1) é um padrão de protocolo para comunicação sem fio a curta distância, desenvolvido pelo *Bluetooth SIG (Special Interest Group)* (BLUETOOTH, 2007). A arquitetura física do *Bluetooth* está composta por duas camadas que são implementadas em hardware: a de enlace RF, para comunicação via canal aéreo entre os dispositivos, e de banda

base, que funciona como controlador de enlace para executar protocolos e rotinas no nível mais baixo da implementação *Bluetooth*, acima da camada RF. A Figura 13 mostra, de forma simplificada, a arquitetura das camadas do sistema *Bluetooth*. Além das camadas anteriores, existem também: o LMP (*Link Management Protocol*), encarregado de administrar a criação, modificação, ou encerramento de enlaces com outros dispositivos; o L2CAP (*Logical Link Control and Adaptation Protocol*), que multiplexa e demultiplexa múltiplos canais através de um enlace lógico compartilhado, e executa tarefas de segmentação e reagrupação de dados; o RFCOM, que é um “protocolo de emulação de cabo serial” que permite conectar um dispositivo serial com a camada L2CAP de um dispositivo *Bluetooth*; e, finalmente, o SDP (*Service Discovery Protocol*), que gerencia recursos para que as aplicações possam descobrir que serviços estão disponíveis nos dispositivos conectados à rede, e determinarem as características de tais serviços.

Figura 13. Estrutura simplificada das diferentes camadas do padrão *Bluetooth*.

5.1.1 Estrutura dos Pacotes enviados através do Canal *Bluetooth*

A função principal da camada banda base no protocolo *Bluetooth* é organizar em pacotes a informação a ser transmitida entre os diferentes pontos de conexão. Tais pacotes possuem um formato bem definido na especificação (BLUETOOTH, 2007), como ilustrado na Figura 14.

Figura 14. Formato do pacote da camada banda base *Bluetooth*.

Os pacotes estão formados por três partes fundamentais: o código de acesso, que pode ser composto por 68 ou 72 bits dependendo do tipo de pacote (Seção 5.1.2), e permite a

identificação individual de cada pacote trocado sobre o enlace físico; o cabeçalho, que contém informação para controle do enlace como o tipo de pacote e informação para correção de erros de transmissão; e, finalmente, a carga útil, que pode conter também um cabeçalho no caso de pacotes assíncronos. Dependendo do tipo de informação a ser transmitida, os pacotes podem conter:

- o código de acesso curto (68 bits), no caso de pacotes de tipo ID. Os pacotes de tipo ID contêm unicamente o ACCESS CODE de 68 bits, e são gerados somente na etapa de estabelecimento da conexão, para identificação dos diferentes dispositivos.
- o código de acesso junto ao cabeçalho.
- o código de acesso, o cabeçalho, e a carga útil.

O conteúdo do código de acesso depende do estágio da transmissão e, em geral, é formado a partir dos 24 bits da parte menos significativa do endereço do dispositivo. Quanto ao cabeçalho do pacote, ele contém informação para controle do enlace e, de forma geral, está composto por cinco campos, distribuídos ao longo de 10 bits, como mostrado na Figura 15. O campo HEC é adicionado pelo módulo gerador de código de correção de erros. O campo TYPE, de tipo de pacote, que é o relevante dentro deste cabeçalho para este caso de estudo, consiste de um código de 4 bits que especifica qual dos 16 diferentes tipos de pacote está em uso (Seção 5.1.2). Informações sobre os demais campos do cabeçalho do pacote podem ser encontrados no manual de especificação do protocolo *Bluetooth* (BLUETOOTH, 2007).

Figura 15. Formato do Cabeçalho do Pacote.

Existe um cabeçalho adicional, correspondente à carga útil do pacote, ilustrado na Figura 16. Nele, o campo LENGTH, que indica o número de bits da carga útil, depende diretamente do campo “TYPE” do cabeçalho do pacote (esta dependência é explicada, de forma mais clara, na Seção 5.3.2). Informações sobre os demais campos do cabeçalho da carga útil podem ser encontrados no manual de especificação do protocolo *Bluetooth* (BLUETOOTH, 2007).

Figura 16. Formato do cabeçalho da carga útil.

5.1.2 Tipos de Pacotes do Protocolo Bluetooth

Existem, no protocolo *Bluetooth*, 16 tipos diferentes de pacotes. O tipo de pacote depende do que é chamado “tipo de enlace lógico”, como também, da quantidade de dados a serem transmitidos. O tipo de enlace entre dois dispositivos *Bluetooth* pode ser de dois tipos:

- enlace ACL (*Asynchronous Connection-Less*): esta é uma conexão de comutação de pacotes que se caracteriza pelo uso de canais virtuais que não possuem reserva de janelas de tempo.
- enlace SCO (*Synchronous Connection-Oriented*): a conexão SCO é uma conexão de comutação de circuitos, concebida para a transmissão de voz, porém com suporte para dados. Esta conexão requer o assinalamento específico de faixas de tempo.

Adicionalmente, existe um tipo de enlace lógico denominado “eSCO” (para taxa de dados estendida), mas, por motivos de implementação da camada banda base, não é de nosso interesse. Para indicar os diferentes tipos de pacotes enviados sobre um enlace lógico, emprega-se o campo de 4 bits, TYPE, do cabeçalho do pacote. Quando se estabelece um enlace, informações de endereço do dispositivo (48 bits), códigos iniciais e valor do sinal de relógio (clock) para sincronização (26 bits), são enviados pelo mestre através de um pacote do tipo FHS. A Tabela 1 apresenta os diferentes tipos de pacote que podem ser enviados através de um enlace ACL ou SCO, e as suas principais características (maiores detalhes de cada tipo podem ser obtidos no manual de especificação do Bluetooth (BLUETOOTH, 2007)).

Tabela 1. Tipos de Pacote *Bluetooth*, segundo código TIPO do cabeçalho, e tipo de enlace.

Código TIPO	Enlace SCO	Enlace ACL	Carga útil	Taxa de transmissão simétrica (Kb/s)	Taxa de transmissão assimétrica (Kb/s)
0000	NULL	NULL	0 bits	0	0
0001	POLL	POLL	0 bits	0	0
0010	FHS	FHS	240 bits	64	64
0011	DM1	DM1	18 bytes	108,8	108,8 - 108,8
0100	N/A	DH1	28 bytes	172,8	172,8 - 172,8
0101	HV1	N/A	240 bits	64	64
0110	HV2	N/A	240 bits	64	64
0111	HV3	N/A	240 bits	64	64
1000	DV	N/A	240 bits	64	64
1001	N/A	AUX1	0 bits	0	0
1010	N/A	DM3	123 bytes	256	384,0 – 54,4
1011	N/A	DH3	185 bytes	384	576,0 – 87,4
1100	eSCO				
1101					
1110	N/A	DM5	226 bytes	286,7	477,8 – 36,3
1111	N/A	DH5	341 bytes	432,6	721 – 57,6

Na presente seção foram tabeladas algumas operações de codificação que são realizadas repetidas vezes sobre as cadeias de bits que compõem os pacotes *Bluetooth*. Na seguinte seção, serão apresentadas com maior precisão tais operações, que, de fato, constituem o fundamento funcional da implementação do controlador *Bluetooth* da Seção 5.1.4.

5.1.3 Operações sobre as Cadeias de Dados

Para serem enviados a um eventual receptor, os pacotes ainda devem atravessar um processo de codificação para aumentar a confiabilidade, segurança, e exclusividade dos dados transmitidos. A Figura 17 mostra o processamento que deve ser realizado sobre o cabeçalho de qualquer pacote *Bluetooth*, transmitido ou recebido. O código HEC é acrescentado para checar a existência de erros originados no processo de transmissão do cabeçalho. Em seguida, os bits do cabeçalho são embaralhados aleatoriamente (*whitening*) para evitar padrões redundantes, e assim, minimizar a polarização DC do pacote. Finalmente, os bits do pacote são protegidos por uma codificação FEC 1/3 para correção de erros de transmissão. O procedimento inverso deve ser aplicado ao pacote quando der sua chegada no receptor.

Figura 17. Operações realizadas sobre o cabeçalho do pacote.

O processamento da carga útil do pacote (Figura 18) é similar àquele do cabeçalho. As únicas diferenças são a utilização de código CRC para detecção de erros, uma função adicional de encriptação (ou decríptação no receptor) para proteção e segurança da informação do usuário, e o uso de codificação FEC 2/3 para correção de erros ocasionados na interface RF. Tanto para o cabeçalho, quanto para a carga útil, as codificações iniciais para o embaralhamento e para o CRC, de sete e oito bits respectivamente, são trocadas entre os dispositivos.

Figura 18. Operações realizadas sobre a carga útil do pacote.

5.1.4 Implementação da Camada Banda Base *Bluetooth*

As operações sobre as cadeias de bits descritas na seção anterior especificam as funções dos módulos da implementação da camada de banda base do protocolo *Bluetooth* realizada pelo Grupo de Projeto de Sistemas Eletrônicos Integrados e Software Aplicado, GSEIS, do Laboratório de Microeletrônica da USP. O grupo GSEIS, como parte do projeto Brazil-IP (BRAZIL, 2004), projetou um controlador de enlace para conexões sem fio do tipo *Bluetooth*. Tal controlador, apresentado na Figura 19, foi implementado na linguagem SystemC no nível RT.

Figura 19. Diagrama hierárquico da implementação da camada banda base Bluetooth.

A implementação do adaptador *Bluetooth* inclui todas as funções necessárias para estabelecer enlaces assíncronos, inclusas na camada banda base da especificação do protocolo *Bluetooth*. O adaptador de *Bluetooth* está hierarquicamente estruturado, como mostrado na Figura 19. Esta hierarquia obedece a quatro níveis de operações realizadas pela camada banda base. O nível inferior corresponde ao tratamento de bits; a seguir, o segundo nível aborda o tratamento de cadeias; o terceiro nível contém as operações de integração das cadeias de bits dentro da temporização do canal de *Bluetooth*; e o nível superior realiza os procedimentos de interface com as camadas superiores e com o transceptor de Radio.

O módulo que encerra a funcionalidade completa da implementação recebe o nome de BASEHW; este módulo comunica-se tanto com a camada de transmissão RF como com camadas superiores de nível lógico. Os módulos SRAM, ACCESSCTR, e PACKETBUFFER realizam, respectivamente, tarefas de armazenamento em memória externa à camada banda base, controle e transmissão de dados entre os módulos SRAM, PACKETBUFFER e RADIOCTR usando os protocolos certos, e armazenamento em memória interna do BASEHW.

A implementação do módulo de banda base inclui dois comportamentos diferentes, um para o modo de operação de transmissão da informação e outro no modo de informação que é passada pelo dispositivo mestre. A operação de transmissão se constitui em uma seqüência de etapas e operações específicas. A configuração da transmissão é iniciada através do mecanismo seguinte:

- 1) escrita, no módulo BASEHW, dos bits de carga útil, do cabeçalho, da configuração do canal e da configuração do transceptor de rádio.
- 2) armazenamento da informação anterior no submódulo SRAM.
- 3) o submódulo ACCESSCTR transfere esta informação para a memória interna do módulo BASEHW, que se encontra no submódulo PACKETBUFFER.
- 4) posteriormente, o submódulo RADIOCTL acessa o PACKETBUFFER e lê os valores da frequência de transmissão e do nível de potência a serem utilizados, além dos bits de configuração específicos para o transceptor de rádio.
- 5) o submódulo PACKETPROC acessa o PACKETBUFFER, para obter a configuração do enlace de *Bluetooth*; esta informação inclui os seguintes indicadores de condição:
 - o enlace está estabelecido, ou não.
 - o enlace é cifrado, ou não, determinado por um flag; em caso de cifragem, a chave de cifragem de 128 bits deve ser carregada.
 - o módulo deve atuar como mestre ou como escravo.
- 6) uma vez que esta informação é obtida pelo PACKETPROC, este permite aos seus módulos internos acessarem o PACKETBUFFER para obter informações de configuração de cada um. O submódulo CORR lê do PACKETBUFFER a palavra de sincronização, enquanto o submódulo STREAMPROC lê a informação do tipo de pacote para determinar quais dos seus submódulos são necessários para a operação de transmissão.
- 7) em seguida, o STREAMPROC gerencia a configuração destes submódulos, dando-lhes acesso ao PACKETBUFFER.

Depois do término da etapa de configuração, começa a etapa de transmissão:

- 1) os pacotes *Bluetooth* passam por operações sobre bits, processados um de cada vez, para daí serem enviados pelo canal sem fio.
- 2) durante a transmissão, o PACKETBUFFER fica ligado ao submódulo STREAMPROC e atua como fonte de dados para a operação.
- 3) os pacotes são transmitidos da seguinte maneira:
 - primeiro, os bits da palavra de sincronização são enviados desde o submódulo CORR.
 - depois, os bits do cabeçalho do pacote são enviados pela saída do submódulo FHEC.
 - finalmente, os bits da carga útil são enviados desde o STREAMPROC.

Similar à transmissão, a operação de recepção é efetuada em uma seqüência de etapas:

- 1) escrita no módulo BASEHW da informação da configuração do canal e da configuração do transceptor de rádio.
- 2) esta informação é transferida através do ACCESSCTR, para o PACKETBUFFER para que seja lida pelos demais submódulos do BASEHW.
- 3) o submódulo RADIOCTR lê a informação da configuração do transceptor de radio, e, depois, o PACKETPROC lê a informação de configuração do enlace.
- 4) quando o PACKETPROC tiver encerrado a fase de configuração, ele permite ao STREAMPROC acessar o PACKETBUFFER para transferir aos seus submódulos a informação de configuração; isso acontece unicamente no caso de o enlace estar previamente estabelecido (caso contrário, não é necessário).

Maiores detalhes da funcionalidade de todos os blocos que compõem as diferentes camadas hierárquicas da implementação do adaptador de rede *Bluetooth* podem ser consultadas no Apêndice A.

5.2 Decodificador de Vídeo *MPEG-4*

MPEG-4 é um padrão ISO/IEC desenvolvido pelo grupo MPEG (*Moving Picture Experts Group*) para compressão de imagens de vídeo digitalizadas previamente. O padrão vem apresentando sucesso em aplicações de TV digital, aplicações gráficas e aplicações multimídia interativas.

O *MPEG-4* oferece um grande e rico conjunto de ferramentas para a codificação de objetos audiovisuais, divididas, principalmente, em *MPEG-4 Systems*, *MPEG-4 Visual* e *MPEG-4 Audio*. Tais conjuntos de ferramentas são utilizados para especificar diferentes aplicações, e, por sua vez, estão divididos em subconjuntos chamados de perfis (*Profiles*) que limitam o conjunto de características que um decodificador deve implementar. Para cada um desses perfis, um ou mais níveis (*Levels*) devem ser definidos, restringindo a complexidade computacional. A abordagem é parecida com o *MPEG-2*, cuja combinação perfil-nível mais conhecida é o perfil principal – nível principal (*Main Profile @ Main Level*). Uma combinação perfil-nível permite que o codificador implemente apenas o subconjunto do padrão que ele precisa. As características do padrão *MPEG-4* incluem (SILVEIRA, 2006):

- compressão de imagens estáticas em níveis de cinzento ou a cores.
- algoritmo baseado na codificação por transformada DCT (*Discrete Cosine Transform*).
- explora as características visuais humanas.
- é mais utilizado como técnica não reversível (*lossy*), ainda que possa ser reversível (*lossless*).
- compromisso entre a qualidade e o fator de compressão.

5.2.1 Formato de Codificação MPEG-4

Segundo a especificação do esquema de codificação, o sistema *MPEG-4* se baseia em uma estrutura hierárquica de objetos, para assim, conseguir processar os dados com eficiência (ISO, 2001). Esta hierarquia é ilustrada pela Figura 20 (SILVEIRA, 2006). Cada um dos objetos que compõem a estrutura hierárquica é explicado resumidamente, a seguir:

Figura 20. Composição Hierárquica dos Objetos de Vídeo MPEG-4.

- Objeto de Vídeo (VO): Um Objeto de Vídeo, ou Objeto Visual (VO), é uma sequência de representações gráficas bidimensionais, cada uma delas, em um ponto diferente no tempo. Após seu processamento, o VO é transmitido pelo codificador na forma de uma cadeia de bits, que inclui a correspondente informação de textura, movimento, e forma. As cadeias de bits de

vários VOs podem ser multiplexadas, de forma que o decodificador receba toda a informação, convertendo-a, assim, em uma cena completa de vídeo decodificada.

- Camada de Objeto de Vídeo (VOL): Camada de informações diversas como textura, movimento, forma, tamanho dos quadros de vídeo, sombras, transparências, etc., de vários VOPs que pertencem ao mesmo VO. O VOL foi projetado especificamente para aumentar a resolução temporal ou espacial de um VO, permitindo tornar os parâmetros anteriores escaláveis.

- Plano de Objeto de Vídeo (VOP): Cada uma das representações gráficas das quais está composto um VO. Existem três tipos diferentes de VOPs dependendo do tipo de informação de codificação que possuam. Primeiro, encontram-se os VOPs intra (I-VOPs), que são codificados sem nenhum tipo de referência a outras imagens; eles informam o ponto da sequência codificada onde a decodificação pode começar, mas contêm apenas uma taxa de compressão moderada. Em segundo lugar estão os VOPs preditivos (P-VOPs), codificados mais eficientemente usando predição de compensação de movimento a partir de um I-VOP ou um P-VOP anterior. Finalmente, os VOPs preditivos bidirecionais (B-VOPs) provêm o maior grau de compressão, mas precisam de VOPs de referência tanto prévios, quanto futuros.

- Macrobloco (MB): Cada imagem (VOP) é dividida em vários macroblocos que, por sua vez, estão compostos por quatro blocos de luminância e dois blocos de crominância. O número de macroblocos dentro de um VOP depende do tamanho e da forma do VOP.

- Bloco: O bloco é a unidade de informação mínima do padrão *MPEG-4* (com a exceção do pixel). Cada bloco é criado através de um processo de amostragem das imagens, e está constituído por oito colunas e oito filas de pixels para um total de 64 pixels, ou 64 bits de coeficientes DCT após o processo de codificação.

- Coeficientes DC e AC: Valores binários resultantes da aplicação da transformada de co-seno a cada bloco. A obtenção destes coeficientes é o ponto de partida para realizar todo o processo de compressão de vídeo *MPEG-4*. (Na Seção 5.3.4, pode ser encontrado o parâmetro “acpred_flag”, o qual define uma das dependências entre os parâmetros do decodificador)

O padrão *MPEG-4* define dois modos fundamentais de codificar as seqüências de imagens que compõem o vídeo a comprimir. O primeiro modo, denominado “intra” (referente à informação contida por dentro de cada imagem), aproveita a redundância espacial inerente a todo tipo de imagem digital. O modo “intra” realiza operações de compressão das cadeias de bits que compõem cada imagem individual de vídeo; assim, é um conjunto de operações

aplicadas fora do contexto temporal, ou seja, sem depender da informação contida em imagens prévias ou futuras.

Adicionalmente, para explorar a redundância temporal na informação das formas (presente entre diferentes imagens), o esquema de codificação inclui o modo “inter”, baseado em algoritmos de compensação e predição de vetores de movimento. De tal modo, é possível comprimir seqüências de vídeo excluindo quadros não relevantes, e incluindo somente a informação relacionada às mudanças entre eles (como exposto na Seção 5.3.4, existe o parâmetro “intra”, que configura este modo de funcionamento na implementação do decodificador *MPEG-4*, e forma parte da definição das dependências entre os parâmetros do sistema). A seguir, são apresentados os aspectos mais relevantes desses modos de codificação, para posteriormente continuar com a descrição da implementação do decodificador *MPEG-4* utilizado.

5.2.1.1 Modo de Codificação “Intra”

Como mencionado anteriormente, este modo de codificação opera sobre imagens estáticas, e independentemente do tempo de cada VOP. De fato, o modo “intra” opera sobre a textura de cada imagem, para assim, diminuir a quantidade de informação necessária a transmitir através de qualquer dos canais (Web – *Streaming Media*, vídeo distribuído em CD, voz, televisão digital) para os quais, está projetado o padrão.

Os VOP codificados no modo “intra” (resultando em I-VOPs) atravessam por uma seqüência específica de passos. De forma simplificada, tais passos se baseiam na realização das seguintes operações:

1. amostragem das imagens de vídeo, e criação das estruturas da Figura 20.
2. cálculo da transformada DCT bidimensional de cada bloco.
3. quantização da matriz de coeficientes DCT de cada bloco. O processo de quantização é efetuado para aproveitar a incapacidade do olho humano para perceber pequenas variações no brilho das cores das frequências maiores do espectro visível (JPEG, 2008). A quantização consiste em dividir, um por um, os coeficientes DCT entre os valores de uma matriz de quantização, e, como resultado, a maior parte dos quocientes dos coeficientes

relacionados a cores de frequências altas será zero. Isto permitirá reduzir significativamente a quantidade de dados de vídeo a processar pelo decodificador.

4. reordenamento dos coeficientes quantificados utilizando algoritmos de mapeamento vertical, horizontal, ou zigue-zague. O resultado de tal reordenamento é um arranjo unidimensional de valores quantificados (Na Seção 5.3.4, pode ser encontrado o parâmetro “dir”, o qual define uma das dependências entre os parâmetros do decodificador).
5. ao primeiro coeficiente, que é o coeficiente DC do bloco, é aplicada uma codificação diferencial. As tabelas para realizar tal codificação encontram-se na especificação do padrão *MPEG-4* (ISO, 2001). Seguidamente, o resto dos coeficientes (coeficientes AC) é codificado em relação ao número de zeros que antecede a cada coeficiente (codificação *Run-Length*).

5.2.1.2 Modo de Codificação “Inter”

A codificação “inter” é utilizada para criar os P-VOPs e B-VOPs. A idéia básica é realizar uma compensação do movimento de um bloco anterior (que pode ser I-VOP ou P-VOP) para um bloco atual P-VOP. Entende-se aqui por compensação de movimento, como um mecanismo de predição das mudanças entre imagens consecutivas.

O modo “inter” aproveita a redundância temporal das imagens em uma sequência de vídeo. Isto significa que, devido a fato de que o conteúdo de duas imagens (ou VOPs, na terminologia *MPEG-4*) sucessivas não muda completamente, não é necessário enviar imagens completas ao decodificador, mas somente a informação da diferença entre elas. Assim, é possível reduzir de maneira importante o tamanho das cadeias de bits que compõem o VOB.

O codificador com compensação de movimento opera da seguinte forma: primeiro, tem-se uma imagem armazenada (imagem de referência) para compará-la com outra imagem, e assim, calcular os denominados vetores de movimento, para várias áreas da imagem. A Figura 21 mostra um exemplo de obtenção do vetor de movimento entre dois blocos a partir do deslocamento entre duas imagens.

Figura 21. Obtenção do vetor de movimento entre os blocos de duas imagens consecutivas.

Segundo a especificação do padrão *MPEG-4*, a resolução de codificação da amplitude destes vetores de movimento pode variar na faixa de um quarto de pixel, meio pixel, e um pixel completo. Assim, para os casos de amplitudes não inteiras ($1/4$ e $1/2$ pixel), a decodificação dos vetores requer um parâmetro de controle de arredondamento para seu cálculo correto. A implementação do decodificador apresentado na Seção 5.2.2 suporta vetores de movimento com resoluções de $1/2$ e 1 pixel (maiores detalhes de compensação de movimento a partir de amostras de vetores não inteiros podem ser consultados na especificação do *MPEG-4* (ISO, 2001)).

Depois da obtenção dos vetores de movimento da imagem atual, a imagem de referência é deslocada de acordo com estes vetores para compensar o movimento entre imagens. Seguidamente, a imagem “predita” resultante é comparada com a imagem atual para produzir um erro de predição chamado de “residual”. O codificador *MPEG-4* possui, por regra, algoritmos para otimização na obtenção do erro residual, que basicamente realizam um particionamento dos macroblocos em tamanhos variáveis, de acordo com a necessidade de maior ou menor granularidade nas diferentes regiões da imagem.

No decodificador, o processo inverso será realizado: a imagem de referência é armazenada, deslocada de acordo com os vetores de movimento recebidos, e, finalmente, o erro residual é somado à imagem deslocada para se recriar a imagem original.

5.2.2 Implementação do Decodificador *MPEG-4*

Existem inúmeras aplicações que incluem decodificadores *MPEG-4*, em software (reprodutores de vídeo e *codecs* para sistemas operacionais) ou hardware (telefones celulares, reprodutores MP4 portáteis, etc). Uma implementação particular de decodificador *MPEG-4* foi realizada no Departamento de Engenharia Elétrica da Universidade Federal de Campina Grande. Este decodificador foi implementado na linguagem SystemC, e, da mesma forma que o controlador banda base *Bluetooth*, foi resultado do projeto Brazil-IP (BRAZIL, 2004).

O padrão de imagem processado por esta implementação tem as características apresentadas a seguir. Cada quadro é composto por 72 linhas e 88 colunas, formando uma matriz de 99 macroblocos de 64 pixels (8x8), ou seja, dispostos em 9 fileiras e 11 colunas.

O decodificador *MPEG-4*, executa todas as operações inversas às do codificador descritas na Seção 5.2.1, decodificando VOBs individuais e obtendo uma sequência de imagens de vídeo descomprimido, adequada para a sua visualização. O diagrama de blocos do decodificador implementado é apresentado na Figura 22. A função dos módulos da implementação, que serviram como casos de estudo no presente trabalho, é explicada, a seguir (os outros componentes podem ser consultados na especificação do *MPEG-4* (ISO, 2001)):

Figura 22. Diagrama de Blocos da Implementação do Decodificador *MPEG-4*.

- CBP (Copiador de Blocos de Predição): O CBP faz a leitura da imagem de referência que se encontra armazenada localmente. Como mencionado na Seção 5.2.1.2, esta imagem de referência, junto com o erro residual, permitirá obter de novo a imagem relacionada aos vetores de movimento recebidos no DVM.
- SI (Scan Inverso): O SI recebe à entrada um arranjo unidimensional e produz à saída um arranjo bidimensional. Já que a matriz quantizada foi reordenada (*scanned*) para se obter uma cadeia de valores unidimensional, o decodificador deve recriar as matrizes originais, de maneira a produzir dados de vídeo em um formato visualizável. O módulo SI efetua o ordenamento inverso da cadeia de bits proveniente do DCDCT; a informação sobre a ordem utilizada para reordenar os dados no codificador está incluída no VOB recebido pelo decodificador *Bitstream*.

Uma descrição completa incluindo todos os blocos da Figura 22 pode ser encontrada no Apêndice B.

5.3 Aplicação da Metodologia de Domínios de Parâmetros aos Modelos do Adaptador *Bluetooth* e o Decodificador *MPEG-4*

A sequência de passos essencial para construir o PDG e o gerador de estímulos correspondente ao DUV foi apresentada nas Seções 4.3 e 4.4. Para o presente trabalho, os modelos que serviram como DUV para aplicação da metodologia de verificação com PDGs foram:

- Controlador *Bluetooth*: STREAMPROC (modo de composição de pacotes), STREAMPROC (modo de decomposição de pacotes), PACKETPROC, e BASEHW.
- Decodificador *MPEG-4*: CBP e SI.

Os outros submódulos da estrutura completa do adaptador *Bluetooth*, e do decodificador *MPEG-4*, não foram utilizados no presente trabalho, dado que seus parâmetros não possuem dependências aplicáveis a metodologia de verificação com PDGs. Nas sub-seções seguintes, mostra-se a especificação dos parâmetros de simulação e a organização de relações de dependência.

5.3.1 Parâmetros dos Módulos do Controlador Banda Base *Bluetooth*

A Tabela 2 apresenta os parâmetros de simulação correspondentes ao módulo STREAMPROC, extraídos do documento da especificação funcional do *Bluetooth* (BLUETOOTH, 2007), e os seus possíveis valores. Todos estes parâmetros são recebidos pelo módulo na sua entrada para realizar as operações sobre as cadeias de bits descritas na Seção 5.1.3, tendo os seguintes conteúdos do sistema:

- W_INIT: código inicial para whitening;
- UAP (Upper Address Part): parte superior do endereço do dispositivo utilizada para código inicial de CRC;
- PAYLD_HEADER: cabeçalho da carga útil;
- PACKET_HEADER: cabeçalho do pacote;
- PACKET_TYPE: tipo do pacote;
- BT_CLOCK: valor do relógio do dispositivo mestre, usado para sincronização;
- BD_ADDRESS: endereço do dispositivo dentro da piconet Bluetooth;
- ENCRYPT: *flag* indicativo de encriptação;
- KC: chave de encriptação;
- PACKET_STREAM: a cadeia de bits correspondente à informação da carga útil, incluído os bits de CRC e FEC;
- PAYLOAD_LENGTH: o comprimento da carga útil, incluído os bits de CRC e FEC.

Tabela 2. Parâmetros de simulação para o STREAMPROC, e os seus respectivos possíveis valores.

Nome do Parâmetro	Valores
W_INIT	0 a 2^7-1
UAP	0 a 2^8-1
PAYLD_HEADER	0 a $2^{16}-1$
PACKET_HEADER	0 a $2^{10}-1$
PACKET_TYPE	0 a 15
BT_CLOCK	0 a $2^{26}-1$
BD_ADDRESS	0 a $2^{48}-1$
ENCRYPT	0 a 1
KC	0 a $2^{128}-1$
PACKET_STREAM	0 a 2745 bits
PAYLD_LGT	0 a 2745

Os módulos PACKETPROC e BASEHW possuem os mesmos parâmetros da Tabela 2, com a adição de mais outros dois, a saber: LAP (*Lower Address Part*), cujos valores podem variar desde 0 até $2^{24}-1$, e MODE, que pode ter os valores 1 ou 2, indicando, respectivamente, se o dispositivo deve transmitir ou receber pacotes de dados.

5.3.2 Definição de Dependências para os Parâmetros do STREAMPROC, PACKETPROC, e BASEHW

A partir da especificação de funções relacionadas a operações sobre cadeias de bits, foi possível estabelecer duas dependências entre os parâmetros apresentados na Tabela 2. A primeira dependência reduz-se à escolha de cifrar ou não os pacotes, isto é, uma dependência do tipo *REQUISITO* entre KC e ENCRYPT. A segunda dependência, entre os parâmetros PACKET_TYPE e PAYLD_LGT, corresponde ao comprimento da cadeia de bits transmitida que deve variar segundo o tipo do pacote que esteja sendo processado no controlador *Bluetooth*. Os demais parâmetros sem dependências não formam subespaços confinados e seus valores são gerados aleatoriamente de forma independente.

Devido à sua extensão e pequena relevância, a descrição das dependências no formato de definição de parâmetros para o *ParaGraph* (que foi escrita manualmente), para os blocos do *Bluetooth*, é apresentada no Apêndice C.

5.3.3 Parâmetros dos Módulos do Decodificador MPEG-4

De maneira similar aos módulos do adaptador *Bluetooth*, os parâmetros para os módulos SI e CBP, e conjunto de valores para cada parâmetro, foram definidos como na Tabela 3. O conteúdo dos mesmos para o sistema é apresentado a seguir:

AC_PRED_FLAG: utilizado para especificar que os blocos foram codificados numa sequência em zigue-zague, e a mesma deve ser adotada na recuperação bidimensional dos coeficientes das imagens;

INTRA: flag indicativo do modo intra (INTRA=1) ou inter (INTRA=0) para cada macrobloco;

DIR: da sequência vertical ou horizontal a ser adotada na recuperação bidimensional dos coeficientes das imagens, para cada macrobloco;

COEFF: vetor de 64 posições para cada macrobloco com a imagem em formato unidimensional;

IMAGE_Y: trata-se um único vetor de 25.344 posições com valores de luminância de todos os pixels de um quadro; na realidade, corresponde a 4 vetores de 64 posições (1 vetor por macrobloco) para cada um dos 99 macroblocos do quadro;

IMAGE_U: trata-se um único vetor de 6.336 posições com valores de cromaância U de todos os pixels de um quadro; na realidade, corresponde a 1 vetor de 64 posições por macrobloco, para cada um dos 99 macroblocos do quadro;

IMAGE_V: como no item anterior, porém para a cromaância V;

MV_X: para cada macrobloco, componente horizontal do vetor de movimento;

MV_Y: para cada macrobloco, componente vertical do vetor de movimento;

RND: parâmetro de configuração do arredondamento no cálculo de vetores de movimento não inteiros, para blocos P-VOP;

Tabela 3. Parâmetros de simulação para os módulos SI e CBP, e os seus respectivos valores.

SI		CBP	
Nome do Parâmetro	Valores	Nome do Parâmetro	Valores
AC_PRED_FLAG	0 a 1	IMAGE_Y	0 a 2^8-1
INTRA	0 a 1	IMAGE_U	0 a 2^8-1
DIR	1 a 2	IMAGE_V	0 a 2^8-1
COEFF	-2048 a 2047	MV_X	-64 a 63
		MV_Y	-64 a 63
		RND	0 a 1
		INTRA	0 a 1

5.3.4 Definição de Dependências para o SI e o CBP

Para o caso de SI, foram descobertas as dependências entre os parâmetros INTRA, AC_PRED_FLAG, e DIR, da seguinte forma: o parâmetro DIR somente terá o efeito de controlar a direção (vertical ou horizontal) de decodificação do ordenamento dos coeficientes DCT quando o parâmetro AC_PRED_FLAG seja igual a '1' (ou seja, há uma dependência de *REQUISITO* entre DIR e AC_PRED_FLAG). O parâmetro AC_PRED_FLAG, por sua vez, servirá para especificar um reordenamento dos coeficientes em zigue-zague para blocos I-VOP, isto é, quando INTRA é igual a '1' (dependência de *REQUISITO* entre AC_PRED_FLAG e INTRA). Para todos os blocos P-VOP (quando INTRA for igual a '0') o coeficientes são decodificação em direção zigue-zague.

Uma última dependência do tipo *CONFLITO* foi detectada entre os parâmetros RND e INTRA do módulo CBP. Tal dependência se cria a partir do fato de que não existem vetores de movimento para blocos do tipo I-VOP (modo inter), ou seja, o parâmetro RND não terá efeito nenhum sobre o circuito quando o parâmetro INTRA seja igual a '1'. Os demais parâmetros sem dependência não formam subespaços confinados e seus valores são gerados aleatoriamente de forma independente.

Devido à sua extensão e pequena relevância, a descrição das dependências no formato de definição de parâmetros para o *ParaGraph*, para os blocos do *MPEG-4* são apresentados no Apêndice C.

6 RESULTADOS

Os dois capítulos anteriores apresentaram, respectivamente, o quadro metodológico da verificação funcional baseada em domínios de parâmetros, e adaptada a fontes de estímulos para um modelo de *testbench* como aquele na Figura 1, na linguagem SystemC, e as características dos núcleos de IP utilizadas como casos de estudo. Este capítulo oferece detalhes quantitativos e qualitativos no que diz respeito à influência que duas classes de estímulos exercem sobre o desempenho da simulação; trata-se da geração de estímulos aleatória tradicional, e da aleatória a partir de domínios de parâmetros, que é o objetivo principal da presente dissertação.

6.1 Metodologia de Simulação de Modelos e Comparação de Resultados

Os modelos apresentados no Capítulo 5 foram simulados empregando tanto geradores de estímulos convencionais (completamente abertos), quanto geradores de estímulos orientados por domínios de parâmetros, sendo que, para realizar as simulações, foi utilizado o modelo de *testbench* ilustrado na Figura 1. A aplicação da metodologia do Capítulo 4 para os casos de estudo, até a obtenção dos geradores de estímulos correspondentes, é detalhado na próxima seção. Para o caso dos módulos do *Bluetooth* a estratégia de cobertura está baseada na medição de número de observações de itens simples (Seção 2.2.1), enquanto que, para os modelos do decodificador *MPEG-4*, são utilizados modelos de cobertura cruzada (Seção 2.2.2). Estes modelos de cobertura foram projetados anteriormente em trabalhos prévios dos

autores relacionados (ROMERO, 2005) (SILVEIRA, 2008), e foram utilizados no presente trabalho sem modificações para aumentar a compatibilidade com os resultados já publicados³.

Primeiramente, todos os modelos de módulos foram simulados estimulando-os com vetores de valores de parâmetros dirigidos com restrições simples, ou seja, restrições que apenas permitem o funcionamento correto da simulação. Seguidamente, os mesmos modelos foram simulados utilizando os geradores criados com ajuda da ferramenta *GET_PRG* introduzida no Capítulo 4.

Ao longo da execução das simulações, diferentes dados foram colhidos, incluindo, tempo de execução de cada transação, progressão da cobertura global, e cobertura individual de eventos; tais dados permitiram realizar uma análise comparativa entre as duas alternativas de estimulação. Todos os dados, resultados, e conclusões, são apresentados em seções seguintes.

6.2 Caracterização do Espaço de Estímulos de Entrada

No capítulo anterior, o conjunto de módulos do adaptador de rede *Bluetooth* e do decodificador MPEG-4 a serem verificados foi listado. Nesta seção apresentamos a aplicação da metodologia descrita no Capítulo 4 e da ferramenta de criação de geradores de estímulos, *GET_PRG*, desenvolvida no âmbito deste trabalho para tais módulos. Para razões de uso eficiente de espaço, mostraremos apenas o caso do módulo SI (scan inverso), do decodificador *MPEG-4*, caso suficientemente ilustrativo da metodologia de redução da redundância dos estímulos através de PDGs. Os resultados obtidos para os outros modelos podem ser consultados nos apêndices C, D e E, no final deste documento.

6.2.1 Definição de Dependências do Submódulo SI

As relações de dependência entre os parâmetros do submódulo SI, mostradas de forma textual na Seção 5.3.4, devem ser definidas segundo a sintaxe para sua respectiva interpretação pela interface *ParaGraph*.

³ A comparação da metodologia de geração de estímulos baseada em domínios de parâmetros com a metodologia tradicional, mantendo esta última na forma original como foi montada e executada, é proposital. Os responsáveis pela verificação, nos trabalhos anteriores, construíram os *testbenches*, e os respectivos geradores de estímulos, de acordo com a sua experiência particular, de uma forma que consideramos como uma tradicional “típica” para efeitos de comparação.

A seguir, o exemplo da especificação das dependências para este submódulo, num arquivo “.def”, é apresentado (O Apêndice C contém a definição das dependências no formato *ParaGraph* para demais submódulos do adaptador de rede Bluetooth, e o submódulo CBP do decodificador *MPEG-4*):

```

DEFINITIONS

    MAP ac_pred_flag:
        0:off;
        1:on;

    MAP intra:
        0:off;
        1:on;

    MAP dir:
        1:one;
        2:two;

    RANGE coeff -2048:+2047;

    REQUIRE dir:
        ac_pred_flag name:on;

    REQUIRE ac_pred_flag:
        intra name:on;

END_DEFINITIONS;

```

6.2.2 Grafo de Domínios de Parâmetros do Submódulo SI

O script Python, que foi apresentado na Seção 4.4, utiliza a função “*read_definitions*” das bibliotecas *ParaGraph*, para interpretar as dependências entre os parâmetros que se encontram especificadas num arquivo de texto “.def” como o acima mostrado. O objeto retornado por tal função é um apontador à classe “*DomainGraph*” que representa o PDG do sistema em questão. A seguir, a Figura 23 mostra o PDG obtido para o caso do submódulo SI (O Apêndice D ilustra os PDGs obtidos para os outros modelos utilizados):

Figura 23. PDG obtido para submódulo SI.

Para o caso do submódulo SI, foram gerados 3 domínios a partir das dependências entre os parâmetros. Neste exemplo, o parâmetro “*coeffs*” é independente dos outros parâmetros, portanto seus valores são gerados aleatoriamente, da forma tradicional.

6.2.3 Obtenção dos Geradores de Estímulos em SystemC

Baseando-se no algoritmo que foi apresentado na Seção 4.4, o *GET_PRG* faz uma varredura do PDG, e gera assim o código SystemC correspondente que contém as restrições de valores dos parâmetros para cada domínio. A seguir o código SystemC obtido para o caso do submódulo SI é mostrado. A apresentação deste código é concisa, portanto, excluem-se seções de código irrelevantes como declarações iniciais, enunciados repetitivos, etc. (O Apêndice E contém os exemplos dos geradores de estímulos obtidos para os outros módulos do *Bluetooth* e o módulo CBP):


```
while(1) {
```

← Começa laço global de geração de estímulos.

```
    ac_pred_flag_distrib.clear();
    intra_distrib.clear();
    direcao_distrib.clear();
    block_coeff_distrib.clear();

    ac_pred_flag_distrib.push(1, 1);
    conf_sptr->ac_pred_flag.set_mode(ac_pred_flag_distrib);

    intra_distrib.push(pair<bool, bool> (1 ,1), 1);
    conf_sptr->intra.set_mode(intra_distrib);

    direcao_distrib.push(1, 1);
    direcao_distrib.push(2, 1);
    direc_sptr->dir.set_mode(direcao_distrib);

    for ( int i=0; i< 64; i++) {
        block_coeff_distrib.push(pair<int, int> (-2048, 2047), 64);
        data_sptr->coeff[i].set_mode(block_coeff_distrib);
    }
```

Modificação dos conjuntos de valores para cada domínio.

```
for (int i=0 ; i<DOMAINSIZE ; i++)
{
    bit_constraint.next();
    qi_constraint.next();
    in_constraint.next();
```

← Começa laço geração de estímulos para um dos domínios.

```
    bit_stimuli_modref.write(new conf_si(bit_constraint.conf_sptr.read()));
    bit_stimuli_driver.write(new conf_si(bit_constraint.conf_sptr.read()));
    qi_stimuli_modref.write(new direction(qi_constraint.direc_sptr.read()));
    qi_stimuli_driver.write(new direction(qi_constraint.direc_sptr.read()));
    in_stimuli_modref.write(new coeffs(in_constraint.data_sptr.read()));
    in_stimuli_driver.write(new coeffs(in_constraint.data_sptr.read()));
```

```
}
```

← Finaliza laço de geração de estímulos do domínio.

```
    .
    .
    .
```

```
}
```

← Continua da mesma maneira para os outros domínios de parâmetros, até a finalização do laço global de geração de estímulos.

6.3 Redução Final do Tamanho do Espaço de Simulação

Os grafos de domínios de parâmetros, ou PDGs, permitem organizar os vetores de entrada ao DUV em domínios de parâmetros, reduzindo o número total de estímulos possíveis que compõem o espaço de simulação. Dependendo das características dos parâmetros de entrada particulares de cada projeto, tal redução variará em proporções diferentes. A Tabela 4 mostra toda a informação relacionada a este tipo de análise do espaço de estímulos de entrada, tanto para o adaptador de banda base *Bluetooth*, quanto para o decodificador *MPEG-4*; estes dados foram obtidos a partir da especificação dos modelos, descritos no capítulo anterior, e da aplicação de funções de medição de cardinalidade dos domínios predefinidas nas bibliotecas *ParaGraph*.

A primeira coluna da Tabela 4 inclui os modelos analisados. As colunas 2 e 3 indicam o correspondente número de parâmetros e dependências entre parâmetros, respectivamente. A coluna 4 mostra o espaço completo de estímulos sem levar em consideração as dependências entre parâmetros, enquanto, nas colunas 5 e 6, mostra-se a redução obtida ao projetar geradores de estímulos de forma manual (ou seja, através de uma metodologia tradicional, completamente aleatória, como em (DA SILVA; MELCHER; ARAUJO, 2004)), e aquela redução que resulta de criar domínios de parâmetros aproveitando as dependências entre eles, respectivamente. A última coluna apresenta a redução relativa entre o espaço de estímulos obtido manualmente, e o espaço gerado utilizando PDGs.

Tabela 4. Relação do tamanho do espaço de parâmetros para os diferentes módulos.

DUV	Número de Parâmetros	Número de Dependências entre Parâmetros	Tamanho do Espaço de Parâmetros (não reduzido)	Tamanho do Espaço de Parâmetros (redução manual)	Tamanho do Espaço de Parâmetros (redução por domínios)	Redução do Espaço de Estímulos (%)
STREAMPROC (Composição)	9	2	6.5E+44	3.1E+43	1.6E+43	50.00
STREAMPROC (Decomposição)	9	2	6.5E+44	3.1E+43	1.6E+43	50.00
PACKETPROC	15	2	6.3E+61	1.4E+59	7.0E+58	50.00
BASEHW	16	2	3.7E+62	1.4E+62	7.1E+61	50.00
MPEG4-SI	4	2	2.1E+06	2.1E+06	1.0E+06	50.00
MPEG4-CBP	7	1	4.2E+24	4.2E+24	3.1E+24	25.00

Segundo a Tabela 4 é possível observar que, embora sejam poucas as dependências existentes entre os parâmetros dos modelos, o aproveitamento de tais dependências pode resultar em

uma redução substancial (do espaço tradicional em relação ao dos domínios) do número de valores de simulação. No caso dos submódulos do *Bluetooth*, a taxa de redução é constante (50%) pelo fato de as dependências entre os parâmetros se manterem as mesmas para todos os níveis da hierarquia do circuito.

6.4 Redundância na Geração de Estímulos

Os PDGs, embora vantajosos para reduzir a redundância inerente aos geradores de estímulos aleatórios, fornecem uma visão apenas qualitativa do conjunto efetivo de estímulos de simulação. Por conseguinte, para conhecer o comportamento real dos geradores de estímulos apresentados na Seção 4.3, realizaram-se simulações, para todos os modelos considerados, registrando-se todos os estímulos provenientes dos geradores. Desta forma, foi possível observar quais dos estímulos encontravam-se dentro ou fora do espaço de valores delimitado pelos domínios de parâmetros.

Inicialmente, os módulos listados na Tabela 5 foram simulados com 200.000 estímulos consecutivos (ou seja, até completar o processamento de 200.000 vetores de teste), sem incluir nenhum tipo de medição da cobertura durante a execução da simulação. É importante notar que, dado que a única intenção destas simulações foi a de contabilizar os casos de teste dentro e fora dos domínios de parâmetros, a inclusão de qualquer critério de cobertura poderia dificultar, ou impossibilitar tal contagem, no caso que um ou mais eventos de cobertura atingisse o número de itens que estabelece o ponto de parada da simulação. Os resultados destas primeiras simulações são mostrados na Tabela 5.

Tabela 5. Estímulos inválidos e tempos de simulação para 200.000 casos de teste.

DUV	Estímulos inválidos do espaço tradicional	Tempo de simulação – tradicional (s)	# Vetores válidos – tradicional (vetores/s)	Tempo de simulação – domínios (s)	# Vetores válidos – domínios (vetores/s)
STREAMPROC (Composição)	99930	32184.60	3.11	31965.40	6.26
STREAMPROC (Decomposição)	100011	21475.74	4.65	34774.88	5.75
PACKETPROC	102122	76680.41	1.28	62293.16	3.21
BASEHW	100154	119233.80	0.84	112034.07	1.79
MPEG4 – SI	99954	901.19	110.01	924.34	216.38
MPEG4 – CBP	50149	4155.23	36.07	4005.72	49.93

Como esperado, o número de estímulos redundantes e/ou inválidos⁴ medidos para cada um dos módulos, na Tabela 5 (coluna 2), tendo como base os 200.000 casos de teste simulados, foi aproximadamente proporcional à taxa de redução apresentada na Tabela 4 (coluna 7). Este resultado também evidencia o correto funcionamento dos geradores de estímulos aleatórios orientados por domínios de parâmetros, criados com ajuda do *GET_PRG* (Seção 4.4). Listados nas colunas 3 e 5 da Tabela 5, estão os tempos de simulação para o caso de estímulos provenientes de geradores aleatórios tradicionais, e para o caso de estímulos gerados a partir de domínios de parâmetros, respectivamente. Finalmente, nas colunas 4 e 6, são mostrados os números de vetores válidos em relação ao tempo de simulação (vetores/segundo) , para vetores gerados da forma tradicional e para vetores baseados em domínios, respectivamente.

Os valores da Tabela 5 nos indicam, então, que para o caso de geração aleatória tradicional, pode haver uma quantidade considerável de estímulos inválidos simulados, significando que se poderia desperdiçar, eventualmente, uma grande proporção do tempo de verificação na simulação de estados desnecessários do sistema⁵.

Em condições ideais (tempo de processamento igual para qualquer tipo de vetor de entrada, tamanho infinito dos canais FIFO entre o gerador de estímulos e os *Driver* e módulo de referência, tempo de espera nulo para os vetores dentro do canal FIFO, etc.) os tempos das colunas 3 e 5, na Tabela 5, deveriam ser idênticos, uma vez que para cada módulo o número de iterações (200.000) manteve-se constante. Entretanto, devido à aleatoriedade nos valores produzidos pelos geradores (por exemplo, com tipos e tamanhos diferentes de pacotes), assim como às seqüências diferentes de valores geradas por cada tipo de fonte de estímulos, houve uma discrepância entre o caso tradicional e o caso de geração baseada em domínios de

⁴ Por motivos de simplicidade, o termo “estímulos redundantes e/ou inválidos” será substituído, daqui em diante, pelo termo “estímulos inválidos”. Analogamente, o termo “estímulos não redundantes e/ou validos” será substituído pelo termo “estímulos válidos”.

⁵ Vale a pena lembrar que estamos tratando de casos onde a metodologia tradicional foi aplicada por engenheiros de verificação em um contexto acadêmico. É importante notar que, com um maior esforço durante a modelagem e a análise da especificação manual do gerador de estímulos tradicional, e acrescentando-se uma experiência maior do engenheiro de verificação, pode-se obter resultados iguais ou mesmo melhores do que manualmente através da metodologia de verificação a partir de domínios de parâmetros. Porém, a metodologia de verificação com PDGs (utilizando a ferramenta apresentada nas Seções 4.3 e 4.4) permite ao engenheiro de verificação planejar, preparar, executar, e avaliar a simulação com um esforço consideravelmente menor do que através de uma metodologia tradicional.

parâmetros. Adicionalmente, existem outros fatores (processos do sistema concorrentes, incerteza das funções de medição de tempo, *overhead* da conexão ao servidor de licenças Synopsys através da rede interna, etc.) que podem afetar o tempo de execução das simulações (YUAN; PIXLEY; AZIZ, 2006).

Independentemente das discrepâncias entre os tempos de execução dos *testbenches* para cada tipo de gerador de estímulos, a metodologia de verificação a partir de PDGs mostra-se notavelmente vantajosa no que diz respeito à taxa de geração de estímulos válidos por unidade de tempo; esta taxa é um indicador importante do desempenho da simulação, já que representa a eficiência dos geradores em termos de velocidade de cobertura dos valores considerados relevantes para o cumprimento da verificação, isto é, os valores que satisfazem os critérios de cobertura funcional. A simulação do submódulo STREAMPROC do adaptador *Bluetooth*, em modo de decomposição de pacotes, foi o caso que apresentou a maior diferença relativa entre os tempos de execução correspondentes a cada tipo de gerador (pela Tabela 5, colunas 3 e 5, a diferença é razoavelmente maior para o caso de geração com PDGs). Não obstante, a taxa de vetores válidos por segundo gerados a partir de domínios de parâmetros é aproximadamente 23% maior do que a taxa de vetores gerados utilizando técnicas tradicionais. Informações mais detalhadas sobre o comportamento estatístico do tempo de processamento dos vetores de teste encontram-se apresentadas no Apêndice F.

6.5 Especificação dos Modelos de Cobertura Funcional

Como já mencionado, os módulos do adaptador de rede *Bluetooth* e do decodificador *MPEG-4* têm diferentes estratégias de cobertura, de acordo com os seus trabalhos originais. Para os módulos do adaptador *Bluetooth*, foi adotada a cobertura por itens simples, cujos eventos são compostos por faixas de valores dos parâmetros. A Tabela 6 mostra todos os eventos de cobertura utilizados para os módulos do *Bluetooth*. Como pode ser observado, como exemplo, os eventos 1, 2, 3 e 4, correspondentes ao parâmetro PACKET_STREAM, são utilizados na cobertura da verificação tanto do STREAMPROC como o PACKETPROC e BASEHW, enquanto os eventos 32, 33, 34, 35, 42 e 43, somente fazem parte da cobertura do PACKETPROC e do BASEHW.

Tabela 6. Modelo de cobertura do *Bluetooth*.

Número de evento de cobertura	Composição		STREAMPROC (Composição e Decomposição)	PACKETPROC e BASEHW
	Parâmetro	Faixas de valores		
1	PACKET_STREAM	0 - 0x3FFFFFFF	X	X
2		0x3FFFFFFF - 0x7FFFFFFF	X	X
3		0x7FFFFFFF - 0xBFFFFFFF	X	X
4		0xBFFFFFFF - 0xFFFFFFFF	X	X
5	PAYLD_LGT	0 - 700	X	X
6		700 - 1400	X	X
7		1400 - 2100	X	X
8		2100 - 2800	X	X
9	Tipo de Pacote: (PACKET_HEADER >> 3) & (0xF)	FHS	X	X
10		DM1	X	X
11		DH1	X	X
12		DM3	X	X
13		DH3	X	X
14		DM5	X	X
15		DH5	X	X
16	PACKET_HEADER	0 - 0xFF	X	X
17		0xFF - 0x1FF	X	X
18		0x1FF - 0x2FF	X	X
19		0x2FF - 0x3FF	X	X
20	LLID e FLOW: (PAY_HEADER) & (0x7)	0 - 2	X	X
21		3 - 4	X	X
22		5 - 6	X	X
23		7 - 8	X	X
24	UAP	0 - 0x3F	X	X
25		0x3F - 0x7F	X	X
26		0x7F - 0xBF	X	X
27		0xBF - 0xFF	X	X
28	W_INIT	0 - 0x3F	X	X
29		0x3F - 0x7F	X	X
30		0x7F - 0xBF	X	X
31		0xBF - 0xFF	X	X
32	LAP	0 - 0x3FFFFFFF		X
33		0x3FFFFFFF - 0x7FFFFFFF		X
34		0x7FFFFFFF - 0xBFFFFFFF		X
35		0xBFFFFFFF - 0xFFFFFFFF		X

36	Kc[0]	0 - 0x3FFFFFF	X	X
37		0x3FFFFFF - 0x7FFFFFF	X	X
38		0x7FFFFFF - 0xBFFFFFF	X	X
39		0xBFFFFFF - 0xFFFFFFFF	X	X
40	ENCRYPT	0	X	X
41		1	X	X
42	MODE	1		X
43		2		X

A Tabela 7 lista todos os eventos que compõem o modelo de cobertura do submódulo SI. Na tabela, os itens dos eventos 1 a 4 são contabilizados unicamente quando os vetores de estímulos gerados coincidirem com a combinação de valores específica de cada um desses eventos de cobertura. Os outros eventos foram especificados pelos autores do modelo com a finalidade de verificar diferentes características funcionais.

Tabela 7. Modelo de cobertura do SI.

Número de evento de cobertura	Composição do evento
1	(INTRA = 1) && (AC_PRED_FLAG = 1) && (DIR = 1)
2	(INTRA = 1) && (AC_PRED_FLAG = 1) && (DIR = 2)
3	(INTRA = 1) && (AC_PRED_FLAG) = 0)
4	(INTRA = 1) && (AC_PRED_FLAG) = 1)
5	INTRA = 0
6	INTRA = 1
7	SCAN_TYPE = 0
8	SCAN_TYPE = 1
9	SCAN_TYPE = 2
10	SCAN_TYPE = 3
11	COEFF = 2047
12	COEFF = -2048
13	COEFF = 0
14	$-1 \geq \text{COEFF} \geq -20$
15	$1 \leq \text{COEFF} \leq 20$

Por último, a Tabela 8 apresenta os eventos de cobertura que compõem o modelo de cobertura do submódulo CBP. Os três primeiros eventos, da mesma forma que os mencionados para o caso do submódulo SI, são contabilizados quando a combinação de estímulos gerados coincidir com os valores especificados por tais eventos. Concluindo, somente o SI e CBP

possuem modelos de cobertura cruzada, os quais, como será mostrado mais à frente, são vantajosos para o desempenho da simulação dos modelos.

Tabela 8. Modelo de cobertura para o CBP.

Número de evento de cobertura	Composição do evento	Número de evento de cobertura	Composição do evento
1	$(RND == 0) \ \&\& \ (INTRA == 1)$	17	$-32 \leq MV_X \leq -16$
2	$(RND == 1) \ \&\& \ (INTRA == 0)$	18	$-16 \leq MV_X \leq -8$
3	$(RND == 0) \ \&\& \ (INTRA == 0)$	19	$-8 \leq MV_X \leq 0$
4	$0 \leq IMAGE_Y \leq 31$	20	$0 \leq MV_X \leq 8$
5	$32 \leq IMAGE_Y \leq 63$	21	$8 \leq MV_X \leq 16$
6	$64 \leq IMAGE_Y \leq 127$	22	$16 \leq MV_X \leq 32$
7	$128 \leq IMAGE_Y \leq 255$	23	$32 \leq MV_X \leq 64$
8	$0 \leq IMAGE_U \leq 31$	24	$-64 \leq MV_Y \leq -32$
9	$32 \leq IMAGE_U \leq 63$	25	$-32 \leq MV_Y \leq -16$
10	$64 \leq IMAGE_U \leq 127$	26	$-16 \leq MV_Y \leq -8$
11	$128 \leq IMAGE_U \leq 255$	27	$-8 \leq MV_Y \leq 0$
12	$0 \leq IMAGE_V \leq 31$	28	$0 \leq MV_Y \leq 8$
13	$32 \leq IMAGE_V \leq 63$	29	$8 \leq MV_Y \leq 16$
14	$64 \leq IMAGE_V \leq 127$	30	$16 \leq MV_Y \leq 32$
15	$128 \leq IMAGE_V \leq 255$	31	$32 \leq MV_Y \leq 64$
16	$-64 \leq MV_X \leq -32$		

6.6 Análise Pós-Simulação da Cobertura Funcional

Tendo como finalidade verificar o impacto do uso da estimulação baseada em domínios de parâmetros sob condições mais realistas do que na Seção 6.4, simulações adicionais utilizando uma estratégia de itens de cobertura funcional foram realizadas. Como critério de completude da verificação definiu-se um número de 10.000 itens para cada evento do modelo de cobertura funcional, seja para a cobertura por itens ou cruzada. A Tabela 9 indica os tempos de simulação medidos para simulações realizadas com os dois tipos de geradores de estímulos. A coluna 2 mostra os tempos de simulação no caso de geração aleatória tradicional, enquanto que a coluna 3 corresponde aos tempos medidos de simulações com geração de estímulos organizados em domínios de parâmetros. A coluna 4 apresenta a relação de redução de tempo de simulação entre as duas abordagens de estimulação aleatória.

Tabela 9. Tempos de simulação sob um modelo de cobertura funcional por itens.

DUV	Tempo de simulação: Geração tradicional (s)	Tempo de simulação: Geração baseada em domínios de parâmetros (s)	Redução no tempo de simulação (%)
StreamProc (Composing)	17,463.56	13,946.46	20.14
StreamProc (Decomposing)	26,457.52	24,440.83	7.62
PacketProc	95,460.19	77,427.76	18.89
BaseHW	174,624.88	145,384.47	16.74
MPEG4 – SI	3,311.26	2,579.57	22.10
MPEG4 – CBP	9,397.90	8,867.25	5.65

Os tempos de simulação medidos foram sempre menores para o caso da estimulação aleatória a partir de domínios de parâmetros. Por exemplo, a cobertura do submódulo SI do decodificador MPEG-4 foi completada em um tempo 22% menor, quando os estímulos foram gerados a partir de domínios de parâmetros, comparados a quando gerados a partir do espaço completo de verificação. Este resultado é bastante significativo, levando-se em conta o fato de que trata-se de um modelo de hardware contendo somente duas dependências simples entre os seus parâmetros de verificação. Portanto, é um resultado que pode ser considerado positivo para a metodologia de verificação com PDGs, e utilizando-se os geradores de estímulos criados com ajuda do *GET_PRG*; é possível que para sistemas com maior quantidade de dependências entre parâmetros a redução do tempo de simulação seja ainda maior.

Em geral, é possível observar que uma estimulação do DUV com um número de valores inválidos reduzido produz uma melhora na evolução global da cobertura funcional. Para demonstrar este fato foram realizadas novas simulações, definindo-se 1000 itens para cada evento de cobertura funcional como critério de finalização da simulação. Este resultado é ilustrado na Figura 24, para o caso do submódulo SI do decodificador *MPEG-4*, na qual o eixo vertical representa o número total de itens necessários para considerar satisfeitos todos os eventos de cobertura funcional definidos.

Figura 24. Evolução global da cobertura do submódulo SI do decodificador *MPEG-4*.

A vantagem essencial do uso de PDGs na verificação dos submódulos do decodificador *MPEG-4* está fundamentada na especificação de modelos de cobertura funcional por combinações de valores dos parâmetros de entrada. Por exemplo, ainda usando-se o caso do submódulo SI, existem três eventos de cobertura cruzada, ou por combinações, no seu modelo de cobertura, a seguir:

```
( rnd == 0 ) && ( intra == 1 )
( rnd == 1 ) && ( intra == 0 )
( rnd == 0 ) && ( intra == 0 )
```

Como mostrado, estes eventos de cobertura requerem geração de valores exatos dos parâmetros de entrada ao DUV para se cumprirem, e, dado que os geradores de estímulos por domínios de parâmetros produzem unicamente combinações de valores que coincidem com este modelo de cobertura, não haverá estímulos descartados durante a contagem dos itens de cobertura. Outra forma de interpretar a situação, é que os eventos de cobertura excluem por sua definição os casos inválidos.

Apesar de os tempos de simulação dos submódulos e do modelo completo do adaptador *Bluetooth* terem sido também menores para o caso da verificação com PDGs, como podem ser vistos na Tabela 9, não existe, no entanto, um modelo de cobertura que considere as mesmas dependências entre os parâmetros de entrada do gerador de estímulos. Na realidade, com isso, para o caso tradicional, mesmo os estímulos inválidos são computados como itens de cobertura, ou seja, se excluídos estes casos de teste, a cobertura de 100% seria atingida somente em um tempo significativamente maior. Como forma de fazer uma checagem do comportamento da evolução de cada evento de cobertura nas simulações dos módulos do adaptador *Bluetooth* (e assim tentar explicar o motivo real da aparente melhoria nos tempos), uma nova simulação foi feita com os modelos, e mediu-se a progressão dos eventos individuais no tempo, como ilustrados da Figura 25 até a Figura 28.

Figura 25. Cobertura individual de itens dos parâmetros com dependências para uma simulação com geração tradicional.

Figura 26. Cobertura individual de itens dos parâmetros sem dependências para uma simulação com geração tradicional.

Figura 27. Cobertura individual de itens dos parâmetros com dependências para uma simulação com geração com domínios de parâmetros.

Os dados representados graficamente nas Figura 25 a Figura 28 foram colhidos no transcurso de simulações para as quais se definiu 1000 itens para cada evento de cobertura funcional como critério de finalização. As figuras mencionadas mostram como exemplo o caso do submódulo STREAMPROC no modo de decomposição de pacotes. A Figura 25, e a Figura 26, correspondem a uma simulação realizada com geradores de estímulos tradicionais, enquanto a Figura 27, e Figura 28, correspondem a simulações realizadas com estímulos gerados a partir de domínios de parâmetros. Pode-se observar pelas Figura 26 e Figura 28, que em ambos os casos (tradicional e com domínios), a execução do *testbench* encerra-se quando o último evento ($2100 \leq \text{PAYLD_LGT} \leq 2800$) atinge a contagem de 1000 itens. De novo, é possível observar que houve uma redução do tempo de simulação (4,5 minutos exatamente), que, entretanto, não está relacionada à redução dos estímulos inválidos do espaço inicial de estímulos, obtida aproveitando-se as dependências entre os parâmetros de entrada dos modelos.

Figura 28. Cobertura individual de itens dos parâmetros sem dependências para uma simulação com geração com domínios de parâmetros.

Comparando-se a Figura 26 com a Figura 28, pode-se concluir que o ganho no tempo de execução da simulação, na verdade, está vinculado aos modos diferentes de funcionamento (com tamanhos de pacotes diferentes) produzidos por diferentes seqüências de vetores de entrada, que conseqüentemente, fazem com que os modelos efetuem tarefas a uma taxa de execução diferente. No caso específico do STREAMPROC, este efeito foi benéfico à metodologia baseada em domínios de parâmetros, porém, isto dependerá do sistema, em particular. Fica evidente, ao se comparar a Figura 25 com a Figura 27, que a metodologia tradicional aparenta ser mais vantajosa, pois a evolução dos diferentes itens de cobertura é aproximadamente duas vezes mais rápida do que no caso de geração a partir de domínios de parâmetros; portanto, este é um resultado da estrutura do PDG para os parâmetros do controlador *Bluetooth*, onde, claramente observa-se que o parâmetro ‘Kc’ somente aparece na metade do número total de domínios.

Os resultados das Seções 6.1, 6.2, 6.3, 6.4, 6.5, e 6.6, foram publicados e apresentados em (CASTRO, 2008) no 7th *IEEE Dallas Circuits And Systems Workshop*.

6.7 Análise Estatística de Vetores de Estímulos

Os tempos de simulação dos modelos listados na Tabela 5 indicam uma possível influência que os diferentes tipos de geradores de estímulos podem exercer, de acordo ao mecanismo e à seqüência de geração de tais estímulos, sobre o funcionamento do sistema verificado. Como já foi mencionado, os dois tipos de geradores de estímulos utilizados no presente trabalho se diferenciam não somente pelo conjunto de valores que conforma seu espaço de simulação, mas também, pela ordem e seqüência de geração dos vetores que contém os valores de cada parâmetro de entrada ao DUV, para cada iteração simulada.

Tentando avaliar o efeito que cada um destes tipos de geração de vetores possa ter sobre o funcionamento do projeto sob verificação, foram realizadas algumas simulações adicionais, desta vez, medindo o tempo de processamento total (do gerador até o módulo de checagem do *testbench*) de todos os estímulos gerados, e ordenando-os de acordo com este tempo de processamento. Adicionalmente, os estímulos foram classificados de acordo com sua origem, assim, podendo corresponder a quatro categorias: estímulos tradicionais totais, estímulos tradicionais válidos, estímulos tradicionais inválidos, e estímulos de domínios de parâmetros (estes sempre válidos).

Dentre todos os submódulos verificados, somente no caso do CBP houve diferenças importantes entre os tempos de simulação de estímulos tradicionais e estímulos gerados a partir de domínios. Após se modificar manualmente o tamanho dos domínios para este caso, e, conseqüentemente, a ordem de geração dos estímulos, foi possível observar uma correspondente alteração na distribuição do tempo de processamento de cada um dos vetores de estimulação. Entendemos que isso é causado pela natureza de funcionamento mesma do CBP, o qual depende da informação processada em etapas prévias.

Em relação aos outros casos analisados, não se apresentaram discrepâncias notáveis nas suas distribuições de tempo dos estímulos. Como exemplo, apresenta-se o histograma de tempos para o BASEHW completo. É possível observar que todos os tipos de estímulos possuem tempos de processamento relativamente semelhantes.

Nas páginas seguintes, apresentam-se os histogramas que representam esta classificação para cada caso estudado. Para maior referência, o Apêndice inclui os histogramas de distribuições de tempos de processamento para os outros modelos estudados.

Distribuições de tempo de cada tipo de estímulo para uma simulação inicial do sub-módulo CBP

Distribuições de tempo de cada tipo de estímulo para uma simulação do sub-módulo CBP, modificando a sequência de geração dos estímulos

Distribuições de tempo de cada tipo de estímulo para o módulo BASEHW

7 CONCLUSÕES E CONSIDERAÇÕES FINAIS

No presente trabalho foi apresentada uma metodologia de verificação baseada em grafos de domínios de parâmetros, adaptada à linguagem SystemC para um modelo de testbench genérico (Figura 1). O formalismo de domínios de parâmetros permite que o espaço de parâmetros de entrada seja otimizado eliminando-se casos de estimulação redundante e inválida. Diferente dos autores de (JERINIĆ, 2005) (JERINIĆ, 2007) (JERINIĆ et al., 2006), a metodologia é utilizada em geração aleatória de estímulos ao invés de estimulação direta. Esta metodologia foi comparada à utilização de uma estratégia de geração convencional de estímulos, como aquela proposta em (DA SILVA; MELCHER; ARAUJO, 2004), e citada em (ROMERO, 2005).

Os resultados de simulações executadas por meio de estímulos provenientes de um espaço de redundância reduzida a partir de PDGs mostram que estes grafos podem ser proveitosos sob algumas situações, mas pouco produtivos sob outras. Em geral:

- os grafos de domínios de parâmetros fornecem um nível de objetividade maior do que um espaço de valores de parâmetros tradicional, no que diz respeito aos vetores de estimulação do DUV. Isto, porque eles permitem visualizar os diferentes conjuntos e combinações aplicadas aos sistemas durante a simulação, e portanto, facilitam a posterior extração de informação relacionada às características e/ou funções relevantes que foram exercitadas.

- estímulos gerados a partir do espaço de domínios de parâmetros ajudam a aumentar a eficiência dos processos de simulação, pois casos redundantes e inválidos são excluídos da execução do testbench.
- estímulos gerados a partir do espaço de domínios de parâmetros ajudam a aumentar a eficiência dos processos de simulação quando as dependências de tais domínios estiverem em concordância com o modelo de cobertura especificado para verificar o sistema (como no caso de cobertura cruzada). Geralmente, é possível relacionar os parâmetros e suas dependências com o modelo de cobertura baseando-se no documento da especificação funcional do sistema.
- na ausência de modelos de cobertura que estejam de acordo com as dependências entre os parâmetros de simulação (como no caso de cobertura por itens), a aplicação de domínios de parâmetros pode parecer pouco produtiva para a verificação, se comparada ao método tradicional. Porém, há de se notar que casos inválidos podem estar sendo computados para a cobertura nesta última estratégia.
- em sistemas cujos parâmetros apresentam dependências insuficientes, ou nulas, a aplicação da metodologia de verificação com PDGs torna-se não vantajosa. No presente trabalho, foram relativamente poucos os modelos de circuitos com aptidão suficiente para demonstrar a influência do uso de dependências entre parâmetros para a sua verificação funcional (3 entre 14 módulos, para o adaptador *Bluetooth*, e 2 entre 10 módulos, para o caso do decodificador *MPEG-4*).
- ferramentas de automatização de código para aplicação da metodologia de verificação com PDGs podem ser implementadas, como no caso do presente trabalho, para criar uma interface entre o ambiente *ParaGraph*, o *testbench*, e a ferramenta de simulação. A ferramenta desenvolvida na linguagem Python, *GET_PRG*, para auxiliar na codificação dos geradores de estímulos baseados em domínios de parâmetros mostra que a automatização do código (neste caso SystemC) para verificação é possível, e, embora não seja possível automatizar a produção de código totalmente, melhorias podem ser incluídas gradualmente até se conseguir ferramentas altamente produtivas.

A verificação de sistemas digitais é uma área de pesquisa que é cada dia mais do interesse das comunidades acadêmica e industrial, por representar um grande desafio na tentativa por aumentar a eficiência do ciclo de produção. Assim, além das propostas apresentadas no presente trabalho, sugere-se que ainda seria vantajoso continuar a pesquisa tanto das técnicas e metodologias estudadas no presente trabalho, quanto de propostas de outros autores. Algumas sugestões para trabalhos futuros na área são:

- expansão das capacidades do ambiente *ParaGraph*, visando aproveitar qualquer tipo de dependência entre os parâmetros de simulação. Atualmente, as bibliotecas fornecidas pelos autores do *ParaGraph* possuem algumas limitações de caráter computacional que não permite a criação de domínios de parâmetros de tamanho ou forma arbitrária.
- dentre os resultados obtidos, considerados positivos para a metodologia de verificação com PDGs, todos foram obtidos a partir de simulações por meio de ambientes de simulação de malha aberta. Portanto, seria interessante considerar a opção de agregar a metodologia atual a outras técnicas de verificação utilizando *testbenches* reativos como, por exemplo, aqueles propostos em (ROMERO, 2005) (HEKMATMPOUR; COULTER, 2003) (LAHBIB et al., 2006).
- estender e testar a metodologia baseada em domínios de parâmetro em outros domínios de aplicação, particularmente, naqueles para os quais parâmetros possuem alto grau de dependência, como o caso de processadores de dados.
- inclusão de características adicionais que melhorem a produtividade de ferramentas como a criada no escopo do presente trabalho, de forma a aumentar a eficiência da simulação, e portanto, da verificação funcional. Estas características incluem: automatização do código dos checadores de cobertura, versatilidade de HVLs, e outras estruturas de organização dos parâmetros de simulação.

REFERÊNCIAS BIBLIOGRÁFICAS

(BENJAMIN et al., 1999) BENJAMIN, M; GEIST, D; HARTMAN, A; WOLFSTHAL, Y; MAS, G; SMEETS, R. **A Study in Coverage-Driven Test Generation**. Proceedings of the 36th ACM/IEEE Conference on Design Automation (DAC 99). New Orleans. Estados Unidos. 1999.

(BERGERON, 2003) BERGERON, J. **Writing Testbenches: Functional Verification of HDL Model, Segunda Edição**. Boston: Kluwer Academic Publishers, 2003.

(BLUETOOTH, 2007) Bluetooth SIG, Inc. **Specification of the Bluetooth System Versão 2.1 + EDR**. 2007. Disponível em <<http://www.bluetooth.com>>. Acesso em outubro de 2007.

(BRAZIL, 2004) BRAZIL-IP Network. 2004. Disponível em <<http://www.brazilip.org.br>>. Acesso em setembro de 2007.

(CASTRO, 2008) CASTRO, C. I.; ROMERO, E. L.; STRUM, M.; CHAU, W. J. “**Automatic Generation Of Random Stimuli Sources Based On Parameter Domains For Functional Verification**”, Proceedings Of The 7th IEEE Dallas Circuits And Systems Workshop, 2008.

(DA SILVA; MELCHER; ARAUJO, 2004) DA SILVA, K.R.G.; MELCHER, E.U.K.; ARAUJO, G. **An Automatic Testbench Generation Tool for a SystemC Functional Verification Methodology**. 17th Symposium on Integrated Circuits and Systems Design, pp. 66 - 70. Porto de Galinhas, PE. Brasil. 2004.

(FERRANDI; RENDINE; SCIUTO, 2002) FERRANDI, F; RENDINE, M; SCIUTO, D. **Functional Verification for SystemC Descriptions Using Constraint Solving**. Proceedings of the 2002 Design, Automation and Test in Europe Conference and Exhibition (DATE.02). Paris. França. 2002.

(FRADKIN; MUCHNIK, 2006) FRADKIN, D.; MUCHNIK, I. **Support Vector Machines for Classification**. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, volume 70, pp. 13-20, 2006.

(FUJITA; GHOSH; MUKUL, 2008) FUJITA, M.; GHOSH, I.; MUKUL, P. **Verification for System Level Design**. Morgan Kaufmann Publishers. Elsevier Inc, 2008.

(GIVARGIS; VAHID, 2000) GIVARGIS, T, VAHID, F. **Parameterized System Design**. Proceedings of IEEE/ACM International Workshop on Hardware/Software Codesign, pp. 98–102. San Diego. Estados Unidos. 2000.

(GRINDAL; OFFUTT; ANDLER, 2004) GRINDAL, M; OFFUTT, J; ANDLER, S. **Combination Testing Strategies: A Survey**. 2004, GMU Technical Report ISE-TR-04-05.

(GUZEY et al., 2008) GUZEY, O.; WANG, L.-C.; LEVITT, J.; FOSTER, H. **Functional Test Selection Based on Unsupervised Support Vector Analysis**. Proceedings of the 45th design automation conference, DAC 2008. Nova Iorque. Estados Unidos. 2008.

(HEKMATMPOUR; COULTER, 2003) HEKMATMPOUR, A; COULTER, J. **Coverage-Directed Management and Optimization of Random Functional verification**. Proceedings of the International Test Conference, 2003, Vol. 1, pp. 148 – 155. Washington, D.C. Estados Unidos.

(ISO, 2001) International Organization for Standarization - ISO. **INTERNATIONAL STANDARD ISO/IEC 14496-2: Information technology - Coding of audio-visual objects - Part 2: Visual**. Segunda Edição. Dezembro, 2001.

(JERINIĆ, 2005) JERINIĆ, V. **PARAGRAPH - Parameter Prüfung für Intellectual Properties**; 2005. 191 pp. Tese de Doutorado - Technische Universität Chemnitz, Alemanha. 14 de junho de 2005.

(JERINIĆ, 2007) JERINIĆ, V. **ParaGraph Overview**. Relatório subministrado pelo autor. 2007.

(JERINIĆ; MÜLLER, 2004) JERINIĆ, V.; MÜLLER, D. **Safe integration of parameterized IP**. INTEGRATION: the VLSI Journal, pp. 193-221. Elsevier. 2004.

(JERINIĆ et al., 2006) JERINIĆ, V.; LANGER, J.; HEINKEL, U.; MIILLER, D. **New Methods and Coverage Metrics for Functional Verification**. Proceedings of the conference on Design, automation and test in Europe (DATE06). pp. 1025-1030. Munique. Alemanha. 2006.

(JPEG, 2008) Joint Photographic Experts Group - JPEG. Disponível em <<http://www.jpeg.org>>. Acesso em dezembro de 2008.

(LAHBIB et al., 2006) LAHBIB, Y.; MISSAOUI, O.; HECHKEL, M.; LAHBIB, D.; MOHAMED-YOSRI, B.; TOURKI, R. **Verification flow optimization using an automatic coverage driven testing policy**. 2006, International Conference on Design and Test of Integrated Systems in Nanoscale Technology, pp. 94 - 99. Tunisia.

(LANGER et al., 2006) LANGER, J.; HEINKEL, U.; JERINIĆ, V.; MÜLLER, D. **Improved Coverage Driven Verification and Corner Case Analysis using Decision Diagrams**. 2006, International Symposium on Communications and Information Technologies, pp. 1179-1184. Bangcoc, Tailândia.

(OPENSPARC, 2009) OpenSPARC. Disponível em <<http://www.sun.com/processors/opensparc>>. Acesso em janeiro de 2009.

(PIXLEY et al., 2003) PIXLEY, C.; CHITTOR, A.; MEYER, F.; MCMASTER, S.; BENUA, D. **Functional Verification 2003: Technology, Tools and Methodology**. Proceedings of the 5th International Conference on ASIC, Vol. 1, pp. 1 - 5. 2003.

(PIZIALI, 2004) PIZIALI, A. **Functional Verification Coverage Measurement and Analysis**. Boston: Kluwer Academic Publishers, 2004.

(PYTHON, 2008) Python Software Foundation. **The Python Programming Language**. Disponível em <<http://www.python.org>>. Acesso em janeiro de 2008.

(REGIMBAL et al., 2003) REGIMBAL, S.; LEMIRE, J. F.; SAVARIA, Y.; BOIS, G.; ABOULHAMID, E.M.; BARON, A. **Automating functional coverage analysis based on an executable specification**. 2003, The 3rd IEEE International Workshop on System-on-Chip for Real-Time Applications, pp. 228 - 234. Calgary, Alberta, Canada.

(ROMERO, 2005) ROMERO, E. L. **Técnicas Para a Verificação Funcional Eficiente de uma Implementação RTL da Camada Banda Base do Protocolo Bluetooth**; 2005. 74 pp. Dissertação de Mestrado - Universidade de São Paulo, Brasil. 2005.

(ROMERO; STRUM; CHAU, 2005) ROMERO, E. L.; STRUM, M.; CHAU, W. J. **Comparing Two Testbench Methods for Hierarchical Functional Verification of a Bluetooth Baseband Adaptor**. Proceedings of the 2005 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS 2005). pp. 327-332. Jersey City, New Jersey, Estados Unidos.

(SILVEIRA, 2006) SILVEIRA, G. S. **RandMovie (Release 1), Especificação do formato MPEG4 Simple Profile Level (0), contendo as características gerais do formato, necessárias para criação do gerador de vídeos randômicos**. Universidade Federal de Campina Grande - UFCG, Centro de Engenharia Elétrica e Informática - CEEI, Departamento de Sistemas e Computação- DSC. Setembro, 2006.

(SILVEIRA, 2008) SILVEIRA, G. S. **Gerador de Vídeo Randômico Sintético para Verificação Funcional de Circuitos Decodificadores de Vídeo MPEG-4**; 2008. 66 páginas. Dissertação de Mestrado - Universidade Federal de Campina Grande. Brasil.

(SU et al., 2006) SU, M.; SHIH, C.; HUANG, J.; JOU, J. **FSM-based transaction-level functional coverage for interface compliance verification**. 2006, Asia and South Pacific Conference on Design Automation, pp 448-453. Yokohama, Japan.

(SYNOPSIS, 2008) Synopsis. OpenVera Website. Disponível em <<http://www.open-vera.com>>. Acesso março de 2008.

(SYSTEMVERILOG, 2008) SystemVerilog. **SystemVerilog Overview**. Disponível em <<http://www.systemverilog.org/overview/overview.html>>. Acesso em março de 2008.

(SYSTEMC, 2008) The Open SystemC Initiative. **SystemC Specification**. Disponível em <<http://www.osci.org>>. Acesso em março de 2008.

(VAPNIK, 1998) VAPNIK, V. **The Nature of Statistical Learning Theory**. Springer-Verlag, 2a edição, 1998.

(VERMA; HARRIS; RAMINENI, 2007) VERMA, S.; HARRIS, I. G.; RAMINENI, K. **Automatic Generation of Functional Coverage Models from CTL**. 2007, High Level Design Validation and Test Workshop, pp. 159-164. Irvine, California.

(WILE; GOSS; ROESNER, 2005) WILE, B.; GOSS, J.; ROESNER, W. **Comprehensive Funcional Verification**. Morgan Kaufmann Publishers. San Francisco : Elsevier Inc, 2005.

(YUAN; PIXLEY; AZIZ, 2006) YUAN, J.; PIXLEY, C.; AZIZ, A. **Constraint-Based Verification**. Springer Science, 2006.

APÊNDICE A

A seguir, é feita uma descrição mais detalhada dos módulos que integram os vários níveis da estrutura hierárquica da implementação do adaptador banda base *Bluetooth*, começando pelo mais baixo.

1. Primeiro Nível

Os submódulos do primeiro nível realizam as operações de encriptação, redução de nível DC (*whitening*), detecção e correção de erros. Tais operações são realizadas sobre os bits da carga útil e os cabeçalhos dos pacotes.

- 1) Submódulo CRC. Este submódulo executa as operações necessárias para a detecção de erros nos bits da carga útil. Os bits da carga útil, lidos do PACKETBUFFER pelo STREAMPROC, são inseridos um a um no submódulo CRC. Um código de 16 bits é calculado com estes bits, utilizando-se um registrador de deslocamento com o polinômio indicado na especificação do protocolo (BLUETOOTH, 2007), para ser, finalmente, acrescentado ao final do pacote. Durante a recepção a totalidade dos bits é inserida no submódulo, incluindo o código previamente calculado na transmissão da unidade remota e, como resultado, um código de 16 bits em zero é obtido quando não forem detectados erros nos bits da carga útil. Quando forem detectados erros nos bits, o código CRC é diferente de zero, sendo ele enviado ao STREAMPROC.
- 2) Submódulo FEC. Submódulo para correção de erros nos bits da carga útil. Durante a transmissão, códigos de redundância são acrescentados aos bits da carga útil neste submódulo. Assim, para cada 10 bits de carga útil, o submódulo FEC acrescenta um código de 5 bits utilizando o esquema FEC 2/3. Para o cálculo dos códigos de redundância, o submódulo FEC recebe do submódulo STREAMPROC os bits da carga

útil. Estes bits são inseridos um a um e, a cada 10, o submódulo produz um código de redundância que é, em seguida, acrescentado à sequência que forma o pacote. O cálculo do código de redundância é feito com um registrador de deslocamento implementando um codificador *Hamming* 10,5 (BLUETOOTH, 2007). Durante a recepção o submódulo FEC tem a função de corrigir os erros produzidos nas seqüências de bits da carga útil dos pacotes de banda base de *Bluetooth*.

- 3) Submódulo FHEC. Neste submódulo se incluem as operações de detecção e correção de erros nos bits do cabeçalho dos pacotes da camada banda base. O comportamento do submódulo FHEC depende da operação que está sendo realizada pelo módulo BASEHW. Quando a operação é de transmissão, o submódulo FHEC tem como função o acréscimo da redundância para detecção e a correção de erros. Para o processo de detecção, o FHEC acrescenta ao cabeçalho de 10 bits um código de redundância calculado com um registrador de deslocamento que implementa o polinômio 647 (representação octal). Na recepção, o submódulo FHEC calcula um código de detecção de erro que é obtido da aplicação do mesmo polinômio utilizado na codificação (transmissão). Se o valor do novo código não for zero, o estado de erro é indicado ao STREAMPROC para a respectiva solicitação de retransmissão às camadas superiores.
- 4) Submódulo WHITE. O submódulo WHITE gera a sequência de bits utilizada para cifrar os pacotes do protocolo *Bluetooth* objetivando reduzir o nível DC. Durante a operação de transmissão o submódulo WHITE é configurado com o valor do relógio de *Bluetooth*. Depois, no momento da transmissão, uma sequência de bits que é utilizada para cifrar o cabeçalho e a carga útil do pacote de banda base é gerada. Esta codificação é realizada pelo submódulo STREAMPROC utilizando-se uma função XOR. O submódulo WHITE é igualmente utilizado durante a operação de recepção, gerando a mesma sequência de bits da transmissão.
- 5) Submódulo CIPHER. Este submódulo implementa a geração da sequência de bits utilizada para cifrar os dados no caso do estabelecimento de enlaces seguros. O submódulo gera uma sequência de bits a partir de certos parâmetros: a chave de enlace (128 bits), o endereço *Bluetooth* completo do dispositivo mestre (48 bits) e o valor do relógio do enlace (26 bits). Após estes valores serem obtidos do PACKETBUFFER, o submódulo CIPHER gera uma sequência de bits que é fundida, no STREAMPROC, aos bits da carga útil do pacote de *Bluetooth* utilizando-se uma função XOR. Durante a recepção, a

codificação é igual à realizada durante a transmissão uma vez que os parâmetros do codificador são previamente comunicados entre os dispositivos *Bluetooth*.

2. Segundo Nível

Como mostra a Figura 19, o segundo nível de hierarquia corresponde aos sub- módulos STREAMPROC, CORR e BTCLOCK. Enquanto os submódulos CORR e STREAMPROC preparam as cadeias dos pacotes, o submódulo BTCLOCK estabelece a referência temporal, isto é, o *clock*, para poder realizar a sincronização do controlador com o canal sem fio.

- 1) Submódulo STREAMPROC. Atua como controlador dos submódulos inferiores de processamento de bits. Ele consiste de uma máquina de estados complexa, que gerencia os submódulos de processamento de bits dependendo do tipo de operação, pacote, e conexão. As operações principais desta máquina são a composição e interpretação de pacotes. A representação gráfica geral das funções do STREAMPROC encontra-se ilustrada através da Figura 17 e da Figura 18.
- 2) Submódulo CORR. Realiza as operações relacionadas ao código de acesso dos pacotes *Bluetooth*. Enquanto o controlador de *Bluetooth* está transmitindo, o submódulo CORR envia o código de acesso ao canal sem fio. Durante a etapa de recepção, o submódulo CORR fica na escuta do canal sem fio para identificar a presença da palavra de sincronização no canal sem fio, dando uma indicação ao PACKETPROC no caso de encontrá-la. Para sua operação, o submódulo CORR lê o código de acesso (64 bits) do PACKETBUFFER e, dependendo do tipo de pacote, acrescenta os bits adicionais de preâmbulo e trailer (BLUETOOTH, 2007) requeridos para aumentar as possibilidades de correlação dos códigos de acesso.
- 3) Submódulo BTCLOCK. O submódulo BTCLOCK atua como a referência de temporização de todo o controlador com respeito ao canal sem fio. Este submódulo é responsável pela geração dos sinais de 1,6 kHz que sincronizam a troca de janelas de frequências e de 1 kHz que controla o envio e recepção de bits pelo canal sem fio.

3. Terceiro Nível

O terceiro nível está composto pelos submódulos PACKETPROC, PACKETBUFFER, SRAM, ACCESSCTR e RADIOCTR, que serão apresentados a seguir.

- 1) Submódulo PACKETPROC. O PACKETPROC se encarrega de sincronizar as operações realizadas sobre cadeias de bits pelo submódulo STREAMPROC, indicando os tempos certos de configuração, geração e interpretação de cadeias de bits.
- 2) Submódulo PACKETBUFFER. É a memória interna de operação do adaptador de rede *Bluetooth*. Este submódulo armazena todas as informações de configuração dos submódulos que se encontram dentro do PACKETPROC; o submódulo tem uma interface pela qual, cada submódulo pode indicar a informação que requer e, em reposta, o PACKETBUFFER envia serialmente a informação requerida.
- 3) Submódulo SRAM. Memória de acesso externo à camada banda base. É utilizada para possibilitar a interação do controlador com as outras camadas do protocolo, sem interromper o processamento do adaptador.
- 4) Submódulo ACCESSCTR. Realiza nos momentos certos as transferências de informação entre os submódulos SRAM e PACKETBUFFER. O submódulo ACCESSCTR indica ao submódulo RADIOCTR quando este pode acessar o PACKETBUFFER para obter os dados de configuração do transceptor de rádio.
- 5) Submódulo RADIOCTR. O RADIOCTR é uma ponte de comunicação entre o transceptor de rádio e a implementação da camada banda base. A interação entre o RADIOCTR e o transceptor de rádio é regida pelos diagramas de tempo encontrados na especificação do transceptor (PYTHON, 2008).

4. Quarto Nível

O último nível está composto unicamente pelo módulo BASEHW, onde toda a implementação da camada banda base do protocolo *Bluetooth* base é reunida.

O módulo BASEHW integra a totalidade da implementação da camada banda base do protocolo *Bluetooth*, este módulo conta com interfaces para se comunicar com as camadas superiores do protocolo e com o transceptor de rádio.

APÊNDICE B

Este apêndice inclui detalhes sobre a implementação do decodificador de vídeo MPEG-4 desenvolvida no Departamento de Engenharia Elétrica da Universidade Federal de Campina Grande, complementares à Seção 5.2.2⁶.

Decodificador de Bitstream (Bitstream)

O decodificador de *bitstream* nada mais é do que um demultiplexador do sistema MPEG4. O *bitstream* é decodificado ao longo do recebimento, se nenhum erro for detectado o *stream* é considerado válido e a decodificação é concluída com o pacote de vídeo. Os erros que podem ser detectados são:

- Recebimento de um VLC (*Variable Length Code*) ilegal;
- Um erro de semântica;
- Mais de 64 coeficientes DCT (*Discrete Cosine Transform*) decodificados em um bloco;
- Cabeçalho de resincronização inconsistente.

Decodificador de Vetores de Movimento (DVM)

Sua função é realizar o escalonamento, decodificação diferencial (somar mediana de três vizinhos). O bit “coded” indica se o vetor de movimento diferencial estava codificado no

⁶ Extraído de: Silveira, George Sobral. *RandMovie (Release 1)*, *Especificação do formato MPEG4 Simple Profile Level (0)*, contendo as características gerais do formato, necessárias para criação do gerador de vídeos randômicos. Universidade Federal de Campina Grande - UFCG, Centro de Engenharia Elétrica e Informática - CEEI, Departamento de Sistemas e Computação- DSC. Setembro, 2006.

bitstream. Caso não, o DVM deve repetir o último vetor. Caso o bit “coded” do primeiro bloco de um macrobloco estiver setado, o DVM deve gerar um vetor de movimento zero.

Decodificação dos Coeficientes DCT (DCDCT)

Sua função é a recomposição das carreiras de zeros que foram suprimidas na codificação.

Quantização Inversa (QI)

Sua função é realizar o cálculo do fator de escalonamento do coeficiente DC (dc_scaler de 6 bits) a partir de quantiser_scale, ou seja, decodificar e, portanto, recuperar a matriz de pixels do sinal original.

- Quantização: Multiplicar os valores oriundos da predição DC/AC pelos valores de escalonamento. Existem valores de escalonamento diferentes para DC e para AC;
- Saturação : Faixa de valores entre [-2048, 2047];

Predição Inversa (PIACDC)

Este bloco realiza o processo de predição para decodificação de coeficientes AC e DC. Este processo é efetuado somente para macroblocos “intra” (pertencentes a I-VOPs) e quando o cabeçalho de vídeo curto é ‘0’. Sob outras condições a predição não é realizada.

Transformada Inversa dos Co-senos (IDCT)

Algoritmo matemático usado na representação da frequência de um bloco de pixels de vídeo. DCT é a transformação discreta reversível e ortogonal entre os domínios de tempo e frequência. Ela poderá ser ‘FDCT’ (Transformada Discreta de Coseno Direta) ou ‘IDCT’ (Transformada Discreta de Coseno Inversa).

Somador (SUM)

Os pixels chegam na mesma sequência nas duas entradas e são somados diretamente. A soma deve ser colocada no endereço correto para que a memória da imagem atual possa servir mais tarde como memória da imagem anterior ao bloco “Copiador de Bloco de Predição”. Simultaneamente ao armazenamento, os pixels de cromaticidade são enviados ao bloco “Conversor YCrCb-RGB”. Após o armazenamento de todos os blocos de um macrobloco são gerados endereços para a leitura dos pixels de luminosidade daquele mesmo bloco para que possam ser enviadas ao bloco “Copiador de Bloco de Predição”. Esta medida diminui os

recursos de armazenamento necessários naquele bloco. Uma RAM externa ao SOMADOR armazena a imagem corrente. No final do frame esta RAM é desconectada do SOMADOR e conectada ao CBP.

Conversor YcrCb-RGB (RGB)

Seu papel é calcular os valores R, G e B a partir dos valores Y, Cr e Cb de cada pixel. Primeiro chegam dois blocos de valores de cromaticidade de um macrobloco. Em seguida, chegam os valores de luminância. Quatro valores de luminância enviados em sequência correspondem a um par de valores de cromaticidade. Os grupos de quatro valores de luminância são enviados numa sequência correspondente à sequência dos valores de cromaticidade enviados antes.

APÊNDICE C

O ambiente para criação de PDGs, *ParaGraph*, possui um formato bem definido que deve ser usado para especificar os parâmetros, valores ou conjuntos de valores, e as dependências entre os parâmetros do projeto sob verificação. A seguir, são apresentados os arquivos no formato *ParaGraph*, para os submódulos do controlador banda base *Bluetooth*, e os módulos do decodificador *MPEG-4*.

Arquivo “*.def*” para os módulos do controlador banda base *Bluetooth*

(Já que os parâmetros, valores, e dependências entre os parâmetros são idênticos para todos os submódulos simulados, apresenta-se somente o arquivo correspondente ao STREAMPROC)

```

DEFINITIONS

    RANGE w_init 0:127;
    RANGE uap 0:255;

    RANGE headers1 0:7;
    RANGE bt_clock 0:67108863;
    RANGE bd_addr 0:16777215;

    DYNAMIC encrypt;
    MAP encrypt:
        0:off;
        1:on;

    RANGE Kc 0:16777215;

    RANGE air_packet 0:16777215;

    DYNAMIC payld_lgt;
    RANGE payld_lgt 16:2712;

    RANGE headers0 16:1023;

    REQUIRE Kc:
        encrypt name:on;

```

```
CONFLICT payld_lgt sec:16:143:
  headers0 sec:16:23;
  headers0 sec:144:151;
  headers0 sec:272:279;
  headers0 sec:400:407;
  headers0 sec:528:535;
  headers0 sec:656:663;
  headers0 sec:784:791;
  headers0 sec:912:919;
```

```
CONFLICT payld_lgt sec:145:2712:
  headers0 sec:16:23;
  headers0 sec:144:151;
  headers0 sec:272:279;
  headers0 sec:400:407;
  headers0 sec:528:535;
  headers0 sec:656:663;
  headers0 sec:784:791;
  headers0 sec:912:919;
```

```
CONFLICT payld_lgt sec:137:2712:
  headers0 sec:24:31;
  headers0 sec:143:151;
  headers0 sec:280:287;
  headers0 sec:408:415;
  headers0 sec:536:543;
  headers0 sec:664:671;
  headers0 sec:792:799;
  headers0 sec:920:927;
```

```
CONFLICT payld_lgt sec:217:2712:
  headers0 sec:32:39;
  headers0 sec:160:167;
  headers0 sec:288:295;
  headers0 sec:416:423;
  headers0 sec:544:551;
  headers0 sec:672:679;
  headers0 sec:800:807;
  headers0 sec:928:935;
```

```
CONFLICT payld_lgt sec:16:135:
  headers0 sec:80:87;
  headers0 sec:208:215;
  headers0 sec:336:343;
  headers0 sec:464:471;
  headers0 sec:592:599;
  headers0 sec:720:727;
  headers0 sec:848:855;
  headers0 sec:976:983;
```

```
CONFLICT payld_lgt sec:969:2712:
  headers0 sec:80:87;
  headers0 sec:208:215;
  headers0 sec:336:343;
  headers0 sec:464:471;
  headers0 sec:592:599;
  headers0 sec:720:727;
  headers0 sec:848:855;
  headers0 sec:976:983;
```

```
CONFLICT payld_lgt sec:16:215:
    headers0 sec:88:95;
    headers0 sec:216:223;
    headers0 sec:344:351;
    headers0 sec:472:579;
    headers0 sec:600:607;
    headers0 sec:728:735;
    headers0 sec:856:863;
    headers0 sec:984:991;

CONFLICT payld_lgt sec:1465:2712:
    headers0 sec:88:95;
    headers0 sec:216:223;
    headers0 sec:344:351;
    headers0 sec:472:579;
    headers0 sec:600:607;
    headers0 sec:728:735;
    headers0 sec:856:863;
    headers0 sec:984:991;

CONFLICT payld_lgt sec:16:967:
    headers0 sec:112:119;
    headers0 sec:240:247;
    headers0 sec:368:375;
    headers0 sec:496:503;
    headers0 sec:624:631;
    headers0 sec:752:759;
    headers0 sec:880:887;
    headers0 sec:1008:1015;

CONFLICT payld_lgt sec:1801:2712:
    headers0 sec:112:119;
    headers0 sec:240:247;
    headers0 sec:368:375;
    headers0 sec:496:503;
    headers0 sec:624:631;
    headers0 sec:752:759;
    headers0 sec:880:887;
    headers0 sec:1008:1015;

CONFLICT payld_lgt sec:16:1463:
    headers0 sec:120:127;
    headers0 sec:248:255;
    headers0 sec:376:383;
    headers0 sec:504:511;
    headers0 sec:632:639;
    headers0 sec:760:767;
    headers0 sec:888:895;
    headers0 sec:1016:1023;
```

```
END_DEFINITIONS;
```

Arquivo “*.def*” para o submódulo CBP do decodificador *MPEG-4*

```
DEFINITIONS

    MAP intra:
        0:off;
        1:on;

    MAP rnd:
        0:off;
        1:on;

    CONFLICT rnd name:1:
        intra name:on;

END_DEFINITIONS;
```

APÊNDICE D

PDG obtido para todos os módulos do controlador *Bluetooth*

Para fins de facilitação da visualização do PDG, os parâmetros sem dependências foram excluídos do grafo. É possível observar que foram gerados 14 domínios a partir das dependências entre os parâmetros.

PDG obtido para o submódulo CBP do decodificador *MPEG-4*

Similarmente, dois domínios foram obtidos para o submódulo CBP. Apesar de serem poucas dependências, obteve-se uma redução substancial nos tempos de simulações pela existência de modelos de cobertura por combinações.

APÊNDICE E

A ferramenta criada sob o presente quadro de trabalho, chamada *GET_PRG*, e escrita na linguagem interpretada Python, mostrou-se proveitosa para assistir na construção de geradores de estímulos na linguagem SystemC, baseados em domínios de parâmetros. O código dos geradores de estímulos, resultantes da aplicação do *GET_PRG*, é apresentado neste apêndice.

Gerador de estímulos a partir de domínios de domínios de parâmetros para os submódulos do controlador banda base *Bluetooth*

```
while(1) {
```

Começa laço global de geração de estímulos.

```
    encrypt_distrib.clear();
    encrypt_distrib.push(pair<int,int> (0 , 0) , 2 );
    trans_sptr->encrypt.set_mode( encrypt_distrib );

    bt_clock_distrib.clear();
    bt_clock_distrib.push(pair<int,int> (0 , 67108863 ) , 67108864 );
    trans_sptr->bt_clock.set_mode( bt_clock_distrib );

    bd_addr_distrib.clear();
    bd_addr_distrib.push(pair<int,int> (0 , 16777215 ) , 16777216 );
    for (int idx = 0 ; idx < AD_LGT ; idx++)
        trans_sptr->bd_addr[ idx ].set_mode( bd_addr_distrib );

    air_packet_distrib.clear();
    air_packet_distrib.push(pair<int,int> (0 , 16777215 ) , 16777216 );
    for (int idx = 0 ; idx < KC_LGT ; idx++)
        trans_sptr->air_packet[ idx ].set_mode( air_packet_distrib );

    w_init_distrib.clear();
    w_init_distrib.push(pair<int,int> (0 , 127 ) , 128 );
    trans_sptr->w_init.set_mode( w_init_distrib );
```

Modificação dos conjuntos de valores para cada domínio.

```

uap_distrib.clear();
uap_distrib.push(pair<int,int> (0 , 255 ) , 256 );
trans_sptr->uap.set_mode( uap_distrib );

payld_lgt_distrib.clear();
payld_lgt_distrib.push(pair<int,int> (144 , 144) , 2 );
trans_sptr->payld_lgt.set_mode( payld_lgt_distrib );

headers1_distrib.clear();
headers1_distrib.push(pair<int,int> (0 , 7 ) , 8 );
trans_sptr->headers(JERINIĆ, 2005).set_mode( headers1_distrib );

headers0_distrib.clear();
headers0_distrib.push(pair<int,int> (16 , 23 ) , 8 );
headers0_distrib.push(pair<int,int> (144 , 151 ) , 8 );
headers0_distrib.push(pair<int,int> (272 , 279 ) , 8 );
headers0_distrib.push(pair<int,int> (400 , 407 ) , 8 );
headers0_distrib.push(pair<int,int> (528 , 535 ) , 8 );
headers0_distrib.push(pair<int,int> (656 , 663 ) , 8 );
headers0_distrib.push(pair<int,int> (784 , 791 ) , 8 );
headers0_distrib.push(pair<int,int> (912 , 919 ) , 8 );
trans_sptr->headers[0].set_mode( headers0_distrib );

for (int domsize = 0 ; domsize < DOMAINSIZEMAX ; domsize++)
{
    if (( trans_stop >= 0 ) && ( trans_counter == trans_stop ))
        modref_verbose = 1;
    else
        modref_verbose = 0;

    trans_constraint.next();

    tr_ptr = new trans_struct(trans_sptr.read());
    tr_ptr2 = new trans_struct(trans_sptr.read());

    tr_ptr->trans_num = trans_counter;
    tr_ptr2->trans_num = trans_counter++;
    tr_ptr->headers[ 1 ] = tr_ptr->headers[ 1 ] + (( tr_ptr->payld_lgt / 8 ) << 3 );

    estimulo_modref.write( tr_ptr2 );
    estimulo_driver.write( tr_ptr );
}

```

Começa laço
geração de
estímulos para um
dos domínios.

Gerador de estímulos a partir de domínios de parâmetros para o submódulo CBP do *MPEG-4*

```

while(1) {

    rnd_distrib.clear();
    intra_distrib.clear();

    intra_distrib.push(1, 2);
    conf_cbp_sptr->intra.set_mode(intra_distrib);

    rnd_distrib.push(0, 2);
    conf_cbp_sptr->rnd.set_mode(rnd_distrib);

```

```

    for (int i = 0 ; i < DOMAINSIZEM ; i++)
    {
        prev_constraint.next();
        prev_vop_sti = prev_constraint.image_yuv_sptr.read();
        prev_stimuli_modref.write(new image_yuv(prev_vop_sti));
        prev_stimuli_driver.write(new image_yuv(prev_vop_sti));

        ifstream mv_sfi("mv.txt");
        mv mv_sti;

        for(int mb=0; mb<MBS_PER_FRAME; mb++) {
            rnd_constraint.next();

            rnd_stimuli_modref.write(new conf_cbp(conf_cbp_sptr.read()));
            rnd_stimuli_driver.write(new conf_cbp(conf_cbp_sptr.read()));

            for(int k=0; k<4; k++) {
                in_constraint.next();
                mv_sti = in_constraint.mv_sptr.read();
                in_stimuli_modref.write(new mv(mv_sti));
                in_stimuli_driver.write(new mv(mv_sti));
            }
        }
    }
}

```

•
•
•

←

Continua da mesma maneira para os outros domínios de parâmetros, até a finalização do loop global de geração de estímulos.

APÊNDICE F

Este apêndice apresenta gráficas adicionais da análise estatística de estímulos aleatórios, complementar à Seção 6.7.

**Distribuições de tempo de cada tipo de estímulo para o sub-módulo STREAMPROC
(composição de pacotes)**

**Distribuições de tempo de cada tipo de estímulo para o sub-módulo STREAMPROC
(decomposição de pacotes)**

Distribuições de tempo de cada tipo de estímulo para o sub-módulo SI